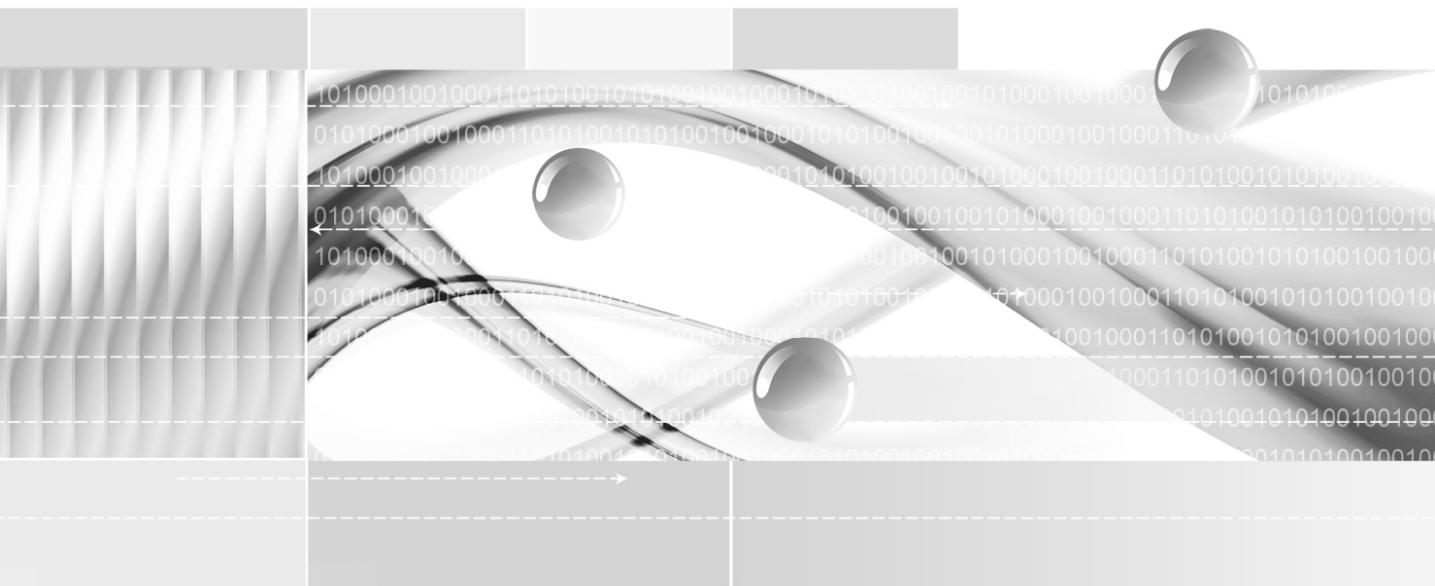


软 件 工 程 系 列 规 划 教 材

面向对象 技术与工具

© 陈文宇 白忠建 吴劲 屈鸿 编著



電子工業出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

本书内容涉及面向对象的思想、方法和技术及两门著名的面向对象程序设计语言——C++语言和 Java 语言。

全书分为两大部分，共 12 章。第一部分介绍面向对象技术，包括：软件工程概述、软件维护、软件工具与集成化环境、面向对象方法、统一建模语言 UML、软件测试；第二部分介绍面向对象程序设计语言，包括：面向对象程序设计语言的核心概念、C++语言实现数据封装、C++语言实现多态性、C++语言实现继承性、Java 语言基础、Java 语言程序设计。

本书是在汲取了国内外有关教材精华的基础上，并结合编者多年面向对象技术和面向对象语言教学经验而编写的，内容注重科学性、先进性、强调实用性。

本书是高等学校软件工程、计算机等相关专业研究生和高年级本科生的教材，也可作为广大工程技术人员和科研人员的参考书。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目（CIP）数据

面向对象技术与工具 / 陈文字，白忠建，吴劲等编著. —北京：电子工业出版社，2008.9

软件工程系列规划教材

ISBN 978-7-121-07051-8

I. 面… II. ①陈… ②白… ③吴… III. 面向对象语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字（2008）第 101659 号

责任编辑：冉 哲

印 刷：

装 订：

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本：787×1092 1/16 印张：33 字数：840 千字

印 次：2008 年 9 月第 1 次印刷

印 数：4000 册 定价：56.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zltz@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

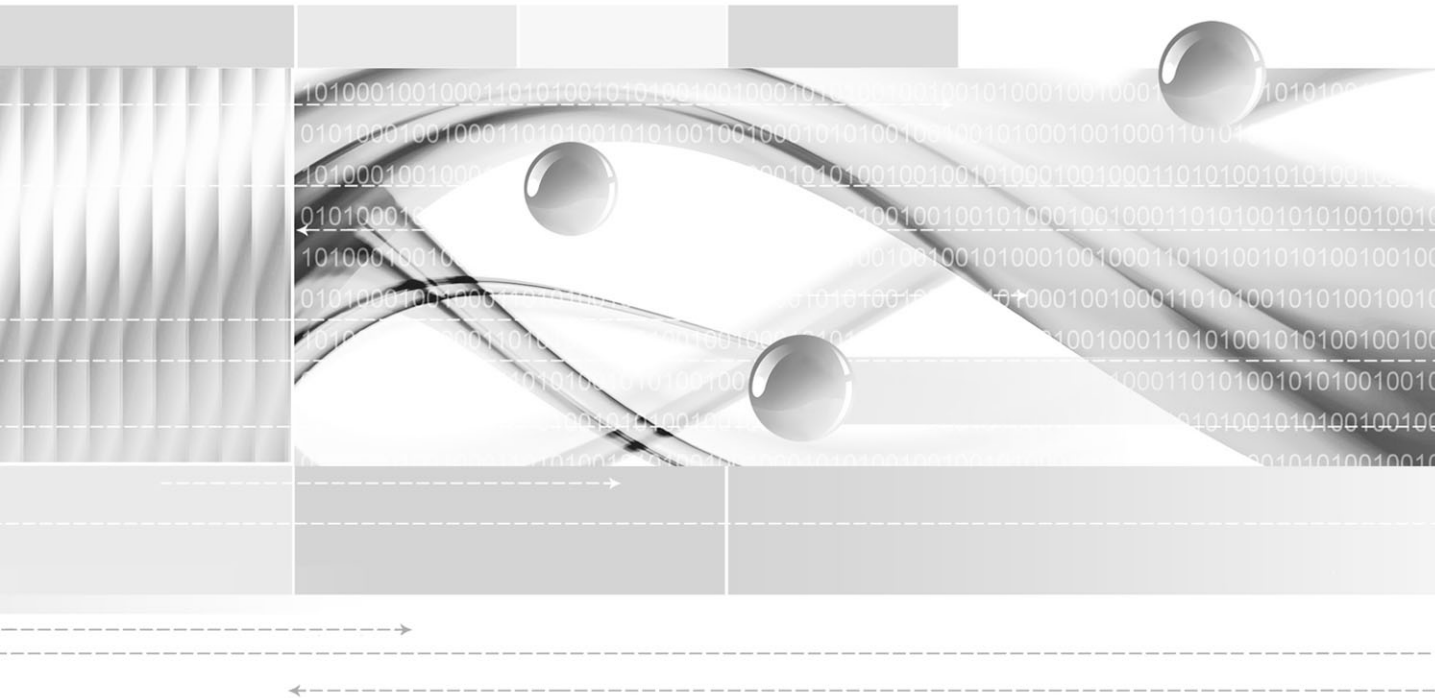
服务热线：（010）88258888。

软件工程系列规划教材 编委会

主任委员：秦志光

副主任委员：徐 谔 任立勇

委 员：张凤荔 桑 楠 邓天权 李雪梅 叶 茂
陆 鑫 刘 玢



出版说明

为适应我国经济结构战略性调整的要求和软件产业发展对人才的迫切需要,实现我国软件人才培养的跨越式发展,教育部和国家发展计划委员会于 2001 年联合批准在国内部分高等学校开办示范性软件学院,努力造就一批进入国际前沿、掌握关键技术、擅长顶层设计的技术带头人,培养一批具备不同专业背景且有市场观念的开发管理、工程管理和软件经营等复合型软件人才,形成一支有相当规模和质量、从事软件开发与应用的专业技术人员队伍。

经过多年的软件人才培养和教学实践,依据国内外企业对软件人才的知识 and 能力需求,以培养高层次、实用型软件人才为目标,我们组织长期从事软件工程硕士教学的专家教授,编写了一套软件工程系列规划教材。该系列教材主要包括《软件开发技术》、《数据库系统及应用》、《操作系统原理与Linux实例设计》、《面向对象技术与工具》及《计算机网络》。本次推出的软件工程系列规划教材内容涵盖软件工程硕士需要掌握的主要知识和基本技术,具有领域宽、实用型强的特点,既可以作为软件学院工程硕士专业基础课教材,也可作为计算机专业高年级本科生和研究生的教材,还可供软件开发和管理人员作为参考书籍。我们还将陆续推出系列教材的习题解答和上机指导及教学用多媒体电子课件,便于教师备课和学生自学,请登录华信教育资源网(<http://www.huaxin.edu.cn>或<http://www.hxedu.com.cn>)注册下载。

在本系列教材的编写过程中,得到了国内众多软件学院的任课教师和软件工程专家的大力支持和帮助,他们提出了许多中肯的意见和建议,对编写工作起到很大的指导作用,对此,编委会和作者表示衷心的感谢!

我们殷切希望本套教材的出版能对国内软件人才的培养起到推波助澜的作用。

尽管我们希望全力以赴编好这套教材,但由于水平和经验有限,难免存在不足和疏漏之处,恳请读者不吝指正。

软件工程系列规划教材编委会

前言

从 20 世纪 80 年代中开始,以 Smalltalk 为代表的面向对象的程序设计语言相继推出,面向对象的方法与技术得到发展,到 90 年代,研究的重点从程序设计语言逐渐转移到面向对象的分析与设计,演化成为一种完整的软件开发方法和系统的技术体系。与此同时,出现了许多面向对象的开发方法的流派,面向对象的方法逐渐成为软件开发的主流。

面向对象的软件开发方法(Object-Oriented Software Development, OOSD)是 20 世纪 80 年代推出的一种全新的软件开发方法。它非常实用而强有力,被誉为 90 年代软件的核心技术之一。

其基本思想是:对问题领域进行自然分割,以更接近人类通常思维的方式建立问题领域的模型,以便对客观的信息实体进行结构和行为的模拟,从而使设计的软件更直接地表现问题的求解过程。

面向对象的软件开发方法以对象作为最基本的元素,是分析和解决问题的核心。

用面向对象方法开发的软件,其结构基于客观世界的对象结构,因此与传统的软件相比,软件本身的内容结构发生了质的变化,因而复用性和扩充性都得到了提高,而且能支持需求的变化。

全书分为两大部分,共 12 章。

第一部分介绍面向对象技术,包括:软件工程概述、软件维护、软件工具与集成化环境、面向对象方法、统一建模语言 UML、软件测试;第二部分介绍面向对象程序设计语言,包括:面向对象程序设计语言的核心概念、C++ 语言实现数据封装、C++ 语言实现多态性、C++ 语言实现继承性、Java 语言基础、Java 语言程序设计。

第 1~6 章由白忠建编写,第 7~10 章由陈文宇和屈鸿编写,第 11、12 章由吴劲编写。

电子科技大学许家珩教授为本教材提供了大量素材并授权使用,在此表示衷心的感谢!

另外,谨对参阅文献的作者和翻译人员表示衷心感谢。

本书在电子科技大学计算机科学与工程学院的徐谡书记和秦志光院长的规划下得以编写和出版。

感谢电子工业出版社的王羽佳编辑,为本书的出版做了大量工作,才使得本书得以同广大读者见面,还要感谢电子科技大学计算机科学与工程学院学生创新中心的大眼睛工作室的同学们,为本书进行的校对和格式修订。

在本书编写过程中,四川汶川发生 8.0 级特大地震,谨以此书告慰逝者,激励生者。

由于作者水平有限,疏漏、欠妥、谬误之处在所难免,敬请读者指正。

编者于电子科技大学
2008 年 6 月

目 录

第 1 章 软件工程概述	(1)
1.1 软件工程的产生和发展	(1)
1.1.1 软件工程的发展过程	(1)
1.1.2 软件危机	(2)
1.1.3 软件工程研究的内容	(3)
1.2 软件与软件工程过程	(4)
1.2.1 软件的概念和特点	(4)
1.2.2 软件工程过程	(6)
1.3 软件过程模型	(6)
1.3.1 瀑布模型	(7)
1.3.2 增量模型	(7)
1.3.3 螺旋模型	(8)
1.3.4 喷泉模型	(9)
1.3.5 智能模型	(9)
1.4 软件开发方法	(10)
1.4.1 结构化开发方法	(10)
1.4.2 原型化开发方法	(11)
1.4.3 面向对象的软件开发方法	(12)
1.5 软件工具与软件开发环境	(13)
习题 1	(14)
第 2 章 软件维护	(15)
2.1 软件维护的基本概念	(15)
2.1.1 软件维护的目的	(15)
2.1.2 软件维护的类型	(15)
2.1.3 软件维护的特性	(17)
2.1.4 软件维护的代价	(18)
2.2 软件维护的过程	(19)
2.3 软件维护技术	(21)
2.4 软件可维护性	(22)
2.4.1 软件可维护性的定义	(22)
2.4.2 提高可维护性的方法	(24)
2.5 逆向工程和再工程	(28)
习题 2	(30)

第 3 章 软件工具与集成化环境	(31)
3.1 软件工具	(31)
3.1.1 软件开发工具	(31)
3.1.2 软件维护工具	(34)
3.1.3 软件管理与支持工具	(35)
3.2 集成化 CASE 环境	(36)
3.2.1 概述	(36)
3.2.2 集成化的 CASE 开发环境的要求	(39)
3.2.3 集成化的 CASE 开发环境的体系结构	(40)
3.3 软件开发工具 Rational Rose	(45)
3.3.1 Rose 工具简介	(45)
3.3.2 业务用例图	(46)
3.3.3 用例图	(47)
3.3.4 类图	(48)
3.3.5 协作图与时序图	(49)
3.3.6 活动图	(50)
3.3.7 状态图	(51)
3.3.8 构件图和部署图	(52)
习题 3	(53)
第 4 章 面向对象方法	(54)
4.1 面向对象方法概述	(54)
4.1.1 什么是面向对象方法	(54)
4.1.2 面向对象方法的主要特点	(55)
4.2 面向对象的基本概念	(56)
4.2.1 对象与类	(56)
4.2.2 继承	(57)
4.2.3 多态性	(58)
4.2.4 消息	(58)
4.3 面向对象的分析	(58)
4.3.1 需求分析中的问题	(59)
4.3.2 OOA 的特点	(60)
4.3.3 OOA 的基本任务与分析过程	(60)
4.4 面向对象的设计	(62)
4.4.1 OOD 的准则	(62)
4.4.2 OOD 的基本任务	(63)
4.5 典型的面向对象方法	(64)
4.5.1 Booch 方法	(64)
4.5.2 Coad/Yourdon 方法	(66)
4.5.3 对象模型技术 OMT	(69)

4.5.4	OOSE 方法	(75)
习题 4		(76)
第 5 章	统一建模语言 (UML)	(77)
5.1	UML 概述	(77)
5.1.1	UML 的形成	(77)
5.1.2	UML 的特点	(78)
5.1.3	UML 建模及其构成	(78)
5.1.4	UML 的图形表示	(80)
5.1.5	通用模型元素	(81)
5.2	建立用例模型	(83)
5.2.1	需求分析与用例建模	(84)
5.2.2	确定执行者	(85)
5.2.3	确定用例	(86)
5.2.4	建立用例之间的关系	(88)
5.2.5	用例建模实例	(89)
5.3	建立静态模型	(92)
5.3.1	类图	(92)
5.3.2	包图	(102)
5.4	建立动态模型	(103)
5.4.1	消息	(103)
5.4.2	状态图	(104)
5.4.3	顺序图	(107)
5.4.4	合作图	(109)
5.4.5	活动图	(111)
5.5	建立实现模型	(114)
5.5.1	构件图	(114)
5.5.2	配置图	(115)
5.6	统一过程及其应用	(117)
5.6.1	UML 与 RUP	(117)
5.6.2	RUP 的特点	(118)
5.6.3	RUP 的二维开发模型	(119)
5.6.4	RUP 的迭代开发模式	(121)
习题 5		(122)
第 6 章	软件测试	(124)
6.1	软件测试概述	(124)
6.1.1	软件测试的基本概念	(124)
6.1.2	软件测试的特点和基本原则	(126)
6.1.3	软件测试过程	(129)
6.1.4	静态分析与动态测试	(131)

6.2	软件测试的策略	(133)
6.2.1	单元测试	(133)
6.2.2	集成测试	(136)
6.2.3	确认测试	(138)
6.2.4	系统测试	(139)
6.2.5	α 测试和 β 测试	(140)
6.2.6	综合测试策略	(141)
6.3	软件调试	(141)
6.3.1	软件调试过程	(142)
6.3.2	软件调试策略	(142)
6.4	面向对象的测试	(144)
6.4.1	面向对象测试的特点	(145)
6.4.2	面向对象测试的类型	(146)
6.4.3	分析模型测试	(148)
6.4.4	面向对象的测试用例	(153)
	习题 6	(154)
第 7 章	面向对象程序设计语言的核心概念	(155)
7.1	面向对象的目标	(155)
7.2	面向对象的核心概念	(157)
7.2.1	数据封装	(157)
7.2.2	继承	(158)
7.2.3	多态性	(159)
7.3	按对象方式思维	(161)
7.4	面向对象的思想和方法	(163)
7.4.1	面向对象是一种认知方法学	(163)
7.4.2	面向对象与软件 IC	(163)
7.4.3	面向对象方法与结构化程序设计方法	(166)
7.4.4	对象是抽象数据类型的实现	(167)
7.5	面向对象的程序设计语言	(168)
第 8 章	C++语言实现数据封装	(173)
8.1	类的定义	(173)
8.2	类的成员	(175)
8.2.1	数据成员	(175)
8.2.2	成员函数	(176)
8.2.3	静态成员	(178)
8.2.4	类外访问成员的方法	(182)
8.3	C++语言的类	(182)
8.4	数据封装和信息隐蔽的意义	(183)
8.5	构造函数	(184)

8.5.1	构造函数的作用	(184)
8.5.2	构造函数的定义	(185)
8.5.3	重载构造函数	(187)
8.6	复制构造函数	(188)
8.7	析构函数	(193)
8.8	对象的创建、释放和初始化	(195)
8.9	对象和指针	(197)
8.9.1	this 指针	(197)
8.9.2	指向类对象的指针	(200)
8.9.3	指向类的成员的指针	(201)
8.10	友元关系	(204)
8.10.1	友元函数	(204)
8.10.2	友元类	(206)
8.10.3	友元关系的总结	(207)
8.11	与类和对象相关的问题	(208)
8.11.1	类类型作为参数类型	(208)
8.11.2	一个类的对象作为另一个类的成员	(210)
8.11.3	临时对象	(211)
	习题 8	(212)
第 9 章	C++语言实现多态性	(213)
9.1	重载运算符	(213)
9.1.1	运算符重载的语法形式	(216)
9.1.2	重载运算符规则	(217)
9.1.3	一元运算符和二元运算符	(219)
9.1.4	重载“++”和“--”的前缀和后缀方式	(227)
9.1.5	重载赋值运算符	(232)
9.1.6	重载运算符“()”和“[]”	(235)
9.1.7	重载输入运算符和输出运算符	(240)
9.1.8	指针悬挂问题	(242)
9.2	C++语言的类型转换	(246)
9.2.1	标准类型转换为类类型	(247)
9.2.2	类类型转换函数	(249)
9.3	实例——复数类重载运算符	(260)
	习题 9	(264)
第 10 章	C++语言实现继承性	(266)
10.1	继承和派生	(266)
10.1.1	为什么要使用继承	(266)
10.1.2	派生类的声明和继承方式	(272)
10.1.3	基类对象的初始化	(281)

10.2	多继承	(288)
10.2.1	多继承的概念	(288)
10.2.2	虚基类	(291)
10.3	继承的意义	(297)
10.3.1	模块的观点	(298)
10.3.2	类型的观点	(298)
10.4	虚函数	(299)
10.4.1	静态多态性	(300)
10.4.2	基类和派生类的指针与对象的关系	(301)
10.4.3	虚函数与多态性	(303)
10.5	纯虚函数和抽象类	(313)
10.6	虚函数实例——Figure 类	(314)
10.7	类属	(321)
10.7.1	无约束类属机制	(321)
10.7.2	约束类属机制	(322)
10.8	模板的概念	(323)
10.8.1	函数模板与模板函数	(323)
10.8.2	类模板与模板类	(326)
10.9	实例——一维数组	(331)
10.10	堆栈、队列的应用	(339)
	习题 10	(342)
第 11 章	Java 语言基础	(344)
11.1	Java 语言的发展历程	(344)
11.2	Java 语言的特点	(346)
11.2.1	简捷性	(346)
11.2.2	面向对象	(346)
11.2.3	动态性	(348)
11.2.4	安全性	(349)
11.2.5	平台无关性和可移植性	(349)
11.2.6	高性能	(349)
11.2.7	多线程	(350)
11.2.8	分布式	(350)
11.2.9	健壮性	(350)
11.3	Java 语言的开发工具包	(351)
11.3.1	JDK 的下载、安装和设置	(351)
11.3.2	JDK 的简介	(352)
11.4	Java 程序的基本结构	(355)
11.5	Java 程序开发实例	(356)
11.5.1	一个简单的 Java Application 程序	(356)
11.5.2	一个简单的 Java Applet 程序	(357)

11.5.3	Java Applet 图形界面的输入/输出	(359)
11.5.4	Java Application 图形界面的输入/输出	(361)
11.6	Java 符号集	(363)
11.7	数据的简单类型	(364)
11.8	常量	(364)
11.9	变量	(365)
11.10	运算符与表达式	(366)
11.10.1	赋值运算与类型转换	(366)
11.10.2	算术运算符	(367)
11.10.3	关系与逻辑运算	(369)
11.10.4	位运算	(370)
11.10.5	其他运算符	(371)
11.10.6	优先级	(371)
11.11	流程控制语句	(371)
11.11.1	分支语句	(372)
11.11.2	循环语句	(375)
11.11.3	跳转语句	(378)
习题 11	(379)
第 12 章	Java 语言程序设计	(380)
12.1	Java 的类和对象	(380)
12.1.1	系统定义的类	(380)
12.1.2	用户程序自定义类	(381)
12.1.3	创建对象与定义构造函数	(383)
12.1.4	类的修饰符	(386)
12.2	域和方法	(386)
12.2.1	域	(386)
12.2.2	方法	(389)
12.3	访问控制符	(392)
12.4	继承	(393)
12.4.1	派生子类	(394)
12.4.2	域的继承与隐藏	(396)
12.4.3	null、this 与 super	(400)
12.5	多态性	(402)
12.5.1	方法的继承	(402)
12.5.2	覆盖	(403)
12.5.3	重载	(403)
12.5.4	构造函数的继承与重载	(406)
12.6	上转型对象	(407)
12.7	接口	(408)
12.7.1	接口的声明	(409)

12.7.2	接口的实现	(410)
12.7.3	接口的回调	(410)
12.7.4	接口作为参数	(411)
12.8	包	(412)
12.8.1	创建包	(413)
12.8.2	包的引用	(413)
12.9	数组	(414)
12.9.1	数组声明	(414)
12.9.2	数组元素的引用及初始化	(415)
12.10	字符串	(417)
12.10.1	String 类	(418)
12.10.2	StringBuffer 类	(421)
12.10.3	Java Application 命令行参数	(423)
12.11	语言基础类库	(423)
12.11.1	Object 类	(423)
12.11.2	数据类型类	(424)
12.11.3	Math 类	(425)
12.11.4	System 类	(425)
12.12	Applet 类与 Applet 程序	(425)
12.12.1	Applet 类	(425)
12.12.2	HTML 文件的参数传递	(428)
12.13	异常处理	(429)
12.13.1	Java 语言异常处理的特点	(429)
12.13.2	异常类的层次	(433)
12.13.3	抛出异常	(435)
12.13.4	异常处理	(436)
12.13.5	嵌套的异常处理	(437)
12.14	Java 多线程机制	(437)
12.14.1	基本概念	(437)
12.14.2	多线程实现方法	(441)
12.15	输入/输出流类	(447)
12.15.1	文件系统	(447)
12.15.2	读/写文件	(452)
12.15.3	抽象流类	(458)
12.15.4	文件输入/输出流类	(459)
12.15.5	加强输入/输出流类	(462)
12.15.6	其他输入/输出流类	(463)
12.15.7	Reader 和 Writer	(470)
12.16	网络编程	(471)
12.16.1	InetAddress 类	(471)

12.16.2	Socket 类和 ServerSocket 类	(473)
12.16.3	DatagramPacket 类和 DatagramSocket 类	(478)
12.16.4	URL 类和 URLConnection 类	(481)
12.17	图形用户界面的设计与实现	(484)
12.17.1	图形用户界面的构成	(484)
12.17.2	布局管理	(485)
12.17.3	组件和事件处理	(491)
12.17.4	Java Swing 基础	(506)
习题 12	(508)
参考文献	(510)

第 1 章 软件工程概述



1.1 软件工程的产生和发展

软件工程（Software Engineering）是在克服 20 世纪 60 年代末所出现的“软件危机”的过程中逐渐形成与发展的。自 1968 年在北大西洋公约组织（NATO）举行的软件可靠性学术会议上正式提出“软件工程”的概念以来，在这 40 年的时间里，软件工程在理论和实践两方面都取得了长足的进步。

软件工程是一门指导计算机软件系统开发和维护的工程学科，是一门新兴的边缘学科，它涉及计算机科学、工程科学、管理科学、数学等多学科。软件工程的研究范围广，不仅包括软件系统的开发方法和技术、管理技术，还包括软件工具、环境及软件开发的规范。

软件是信息化的核心，国民经济、国防建设、社会发展及人民生活都离不开软件。软件产业关系到国家经济发展和文化安全，体现了国家综合实力，是决定 21 世纪国际竞争地位的战略产业。因此，大力推广应用软件工程的开发技术及管理技术，提高软件工程的应用水平，对促进我国软件产业与国际接轨，推动软件产业的迅速发展起着十分重要的作用。



1.1.1 软件工程的发展过程

软件工程的产生和发展是与软件的发展过程紧密相关的。自从第一台电子计算机诞生以来，就开始了软件的生产，“软件工程”提出至今，它的发展经历了 4 个重要阶段。

1. 第一代软件工程

20 世纪 60 年代末，软件生产主要采用“生产作坊方式”。随着软件需求量、规模及复杂度的迅速增大，生产作坊的方式已不能够适应软件生产的需要，出现了所谓“软件危机”（Software Crisis），其表现为软件生产效率低，大量质量低劣的软件涌入市场或在开发过程中夭折。软件危机不断扩大，对软件生产已经产生了严重危害。

为了克服软件危机，在 NATO 举行的软件可靠性会议上第一次提出“软件工程”这一名词，将软件开发纳入了工程化的轨道，基本形成了软件工程的概念、框架、技术和方法。这阶段又称为**传统的软件工程**。

2. 第二代软件工程

从 20 世纪 80 年代中期开始，以 Smalltalk 为代表的面向对象的程序设计语言推出，面向对象的方法与技术得到发展。从 20 世纪 90 年代起，研究的重点从程序设计语言逐渐转移到面向对象的分析与设计，演化成为一种完整的软件开发方法和系统的技术体系。与此同时，



出现了许多面向对象的开发方法的流派，面向对象的方法逐渐成为软件开发的主流。所以这一阶段又称为**对象工程**。

3. 第三代软件工程

随着软件规模的不断增大，开发人员也随之增多，开发时间相应持续增长，加之软件是知识密集型的逻辑思维产品，这些都增加了软件工程管理的难度。人们在软件开发的实践过程中认识到：提高软件生产率，保证软件质量的关键是对“软件过程”的控制和管理，是软件开发和维护中的管理和支持能力。提出对软件项目管理的计划、组织、成本估算、质量保证、软件配置管理等技术与策略，逐步形成了**软件过程工程**。

4. 第四代软件工程

从 20 世纪 90 年代起，基于构件（Component）的开发方法取得重要进展，软件系统的开发可通过使用现存的可复用构件组装完成，而无须从头开始构造，从而达到提高效率和质量，降低成本的目的。软件复用技术及构件技术的发展，对克服软件危机提供了一条有效途径。将这一阶段称为**构件工程**。

1.1.2 软件危机

1. 软件危机的产生

“软件危机”（Software Crisis）的出现是由于软件的规模越来越大，复杂度不断增大，而软件需求量也不断增大，“生产作坊式”的软件开发模式及技术已不能满足软件发展的需要。

软件开发过程是一种高密集度的脑力劳动，需要投入大量的人力、物力和财力，如果软件开发的模式及技术不能适应软件发展的需要，将致使大量质量低劣的软件产品涌向市场，有的甚至在开发过程中就夭折了。国外在开发一些大型软件系统时，遇到了许多困难，有的系统最终彻底失败了；有的系统则比原计划推迟了好多年，而且费用大大超过了预算；有的系统不能符合用户当初的期望；有的系统则无法进行修改维护。典型的例子如下。

IBM 公司的 OS/360，共约 100 万条指令，花费了 5000 人年，经费达数亿美元，而结果却令人沮丧，错误多达 2000 个以上，系统根本无法正常运行。OS/360 系统的负责人 Brooks 这样描述开发过程的困难和混乱：“就像野兽在泥潭中做垂死挣扎，挣扎得越猛，陷得越深，最后没有一个野兽能够逃脱淹没在泥潭中的命运……”

1967 年，前苏联“联盟一号”载人宇宙飞船，由于其软件设计时忽略了一个小数点，导致返航时打不开降落伞，当进入大气层时因摩擦力太大而烧毁，造成机毁人亡的巨大损失。

还有，可以称为 20 世纪世界上最精心设计，并花费了巨额投资的美国阿波罗登月飞行计划使用的软件，也仍然没有避免出错，例如，阿波罗 8 号由于太空飞船的一个计算机软件错误，造成存储器的一部分信息丢失；阿波罗 14 号在飞行的 10 天中，出现了 18 个软件错误。

2. 软件危机的表现

20 世纪 60 年代末期所发生的软件危机，反映在软件可靠性没有保障、软件维护工作量

大、费用不断上升、进度无法预测、成本增长无法控制、程序人员无限度增加等方面，以致形成人们难以控制软件开发的局面。

软件危机主要表现在以下两个方面。

- ① 软件产品质量低劣，甚至在开发过程中就夭折。
- ② 软件生产率低，不能满足需要。

软件危机所造成的严重后果已致使世界各国的软件产业危机四伏，面临崩溃，克服软件危机刻不容缓。从 NATO 会议以来，世界各国的软件工作者为克服软件危机进行了许多开创性的工作，在软件工程的理论研究和工程实践两个方面都取得长足的进步，缓解了软件危机。但距离实现彻底克服软件危机这个软件工程的最终目标，仍然任重而道远，还需要软件工作者付出长期艰苦的努力。

1.1.3 软件工程研究的内容

1. 软件工程的定义

自从 1968 年提出软件工程这个名词，对于软件工程就有了各种各样的定义，但是它们的基本思想都是强调在软件开发过程中应用工程化原则的重要性。

1983 年，IEEE（美国电气与电子工程师协会）所下的定义是：软件工程是开发、运行、维护和修复软件的系统方法。

1990 年，IEEE 又将定义更改为：对软件开发、运作、维护的系统化的、有规范的、可定量的方法之应用，即对软件的工程化应用。

从软件工程的定义可见，软件工程是一门指导软件开发的工程学科，它以计算机理论及其他相关学科的理论为指导，采用工程化的概念、原理、技术和方法进行软件的开发和维护，把经过实践证明的科学的管理措施与最先进的技术方法结合起来。软件工程研究的目标是“以较少的投资获取高质量的软件”。

2. 软件工程研究的内容

软件工程有方法、工具和过程三个要素。软件工程方法研究软件开发是“如何做”的。软件工具是研究支撑软件开发方法的工具、为方法的运用提供自动或者半自动的支撑环境。软件工具的集成环境，又称为计算机辅助软件工程（Computer-Aided Software Engineering, CASE）。软件工程过程则是指将软件工程方法与软件工具相结合，实现合理、及时地进行软件开发的目标，为开发出高质量软件而规定各项任务的工作步骤。

软件工程是一门新兴的边缘学科，涉及的学科多，研究的范围广。归结起来，软件工程研究的主要是以下 4 个方面的内容。

① 方法与技术。软件开发方法主要讨论软件开发的各种方法及其工作模型，包括多方面的任务，如软件系统需求分析、总体设计，以及如何构建良好的软件结构，数据结构及算法设计等，同时讨论具体实现的技术。

② 工具及环境。软件工具为软件工程方法提供了支持，研究计算机辅助软件工程，建立软件工程环境。



③ 软件工程管理。软件工程管理是指对软件工程全过程的控制和管理,包括计划安排、成本估算、项目管理、软件质量管理。

④ 标准与规范。软件的标准化与规范化,使得各项工作有章可循,以保证软件生产率和软件质量的提高。软件工程标准可分为:国际标准、行业标准、企业规范和项目规范。

必须强调的是,随着人们对软件系统研究的逐渐深入,软件工程所研究的内容也不是一成不变的。

3. 软件工程的基本原则

过去,软件工程的基本原则是抽象化、模块化、清晰的结构、精确的设计规格说明。但是,今天的认识已经发生了很大的变化。现在提出的软件工程四条基本原则如下。

第一,必须认识软件需求的变动性,以便采取适当措施来保证产品能更好地满足用户要求。在软件设计中,通常要考虑模块化、抽象与信息隐蔽、局部化、一致性等原则。

第二,稳妥的设计方法大大地方便软件开发,以达到软件工程的目标。软件工具与环境对软件设计的支持十分重要。

第三,软件工程项目的质量与经济开销直接取决于为该工程提供的支撑的质量与效用。

第四,只有在强调对软件过程进行有效管理的情况下,才能实现有效的软件工程。



1.2 软件与软件工程过程

“软件工程”是在软件生产中采用工程化的方法,采用一系列科学的、现代化的方法技术来开发软件。这种工程化的思想贯穿了软件开发和维护的全过程。

为了进一步学习有关软件工程的方法和技术,先介绍软件、软件生存期及软件工程过程这几个重要的概念。



1.2.1 软件的概念和特点

1. 软件 (Software)

“软件就是程序,开发软件就是编写程序”,这是一个错误的观念。这种错误观点的长期存在,影响了软件工程的正常发展。

事实上,正如 Boehm 指出的:软件是程序及开发、使用和维护程序所需的所有文档。它由应用程序、系统程序、面向用户的文档和面向开发者的文档 4 部分构成。

2. 软件的特点

- ① 软件是一种逻辑实体,不是具体的物理实体。
- ② 软件产品的生产主要是研制。
- ③ 软件具有“复杂性”,其开发和运行常受到计算机系统的限制。
- ④ 软件成本昂贵,其开发方式目前尚未完全摆脱手工生产方式。
- ⑤ 软件不存在磨损和老化问题,但存在退化问题。

图 1-1 给出了硬件失效率曲线,它是一个 U 形曲线(即浴盆曲线),这表明硬件随着使

用时间的增加，失效率急剧上升。

图 1-2 描述了软件失效率曲线，它没有 U 形曲线的右半翼，这表明软件随着使用时间的增加，失效率降低，因为软件不存在磨损和老化问题，然而存在退化问题。

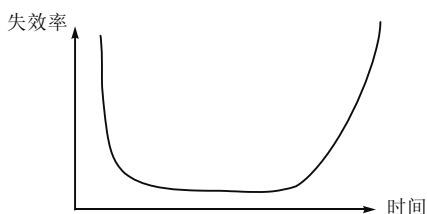


图 1-1 硬件失效率曲线

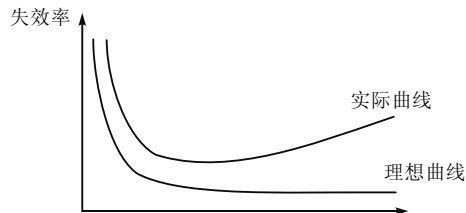


图 1-2 软件失效率曲线

3. 软件生存期

软件生命周期（SDLD）是指一个从用户需求开始，经过开发、交付使用，在使用中不断地增补修订，直至软件报废的全过程，亦称软件生存期（Life Cycle）。

GB 8567 中规定，软件生命周期分为 7 个阶段。

（1）可行性研究和项目开发计划

可行性研究和项目开发计划阶段必须要回答的问题是“要解决的问题是什么”。

（2）需求分析

需求分析阶段的任务不是具体地解决问题，而是准确地确定“软件系统必须做什么”，确定软件系统必须具备哪些功能。

（3）概要设计

概要设计就是设计软件的结构，该结构由哪些模块组成，这些模块的层次结构是怎样的，这些模块的调用关系是怎样的，每个模块的功能是什么。同时还要设计该项目的应用系统的总体数据结构和数据库结构，即应用系统要存储什么数据，这些数据是什么样的结构，它们之间有什么关系等。

（4）详细设计

详细设计阶段就是为每个模块完成的功能进行具体描述，要把功能描述变为精确的、结构化的过程描述。

（5）编码

编码阶段就是把每个模块的控制结构转换成计算机可接受的程序代码，即写成以某特定程序设计语言表示的“源程序清单”。

（6）测试

测试是保证软件质量的重要手段，其主要方式是在设计测试用例的基础上检验软件的各个组成部分。测试分为模块测试、组装测试和确认测试。

（7）维护

软件维护是软件生存期中时间最长的阶段。已交付的软件投入正式使用后，便进入软件维护阶段，它可以持续几年甚至几十年。



在大部分文献中将生存周期划分为 5 个阶段，即要求定义、设计、编码、测试及维护。其中要求定义阶段包括可行性研究和项目开发计划、需求分析，设计阶段包括概要设计和详细设计。

为了描述软件生存期的活动，提出了多种生存期模型，例如：瀑布模型、循环模型、演化模型、螺旋模型等。

1.2.2 软件工程过程

按照 IEEE 计算机学会和 ACM 联合推出的软件工程知识体系（Software Engineering Body Knowledge, SWEBOK），将软件工程过程（Software Engineering Process）知识域（Knowledge Areas）划分为 6 个知识子域：基本概念、过程基础设施、过程度量、过程定义、定性分析和过程实施与变更。也就是说，软件工程过程是指在软件工具的支持下，所进行的一系列软件工程活动。通常包括以下 4 类基本过程。

- ① 软件规格说明：规定软件的功能及其运行环境。
- ② 软件开发：产生满足规格说明的软件。
- ③ 软件确认：确认软件能够完成客户提出的要求。
- ④ 软件演进：为满足客户的变更要求，软件必须在使用 的过程中演进。

软件工程过程具有可理解性、可见性（过程的进展和结果可见）、可靠性、可支持性（易于使用 CASE 工具支持）、可维护性、可接受性（为软件工程师的接受）、开发效率和健壮性（抵御外部意外错误的能力）等特性。

如图 1-3 所示，在软件工程的三要素中，软件工程过程将人员、方法与规范、工具和管理有机地结合起来，形成一个能有效控制软件开发质量的运行机制。

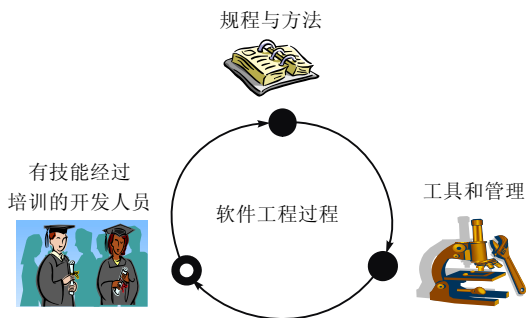


图 1-3 软件工程过程



1.3 软件过程模型

软件过程模型也称为软件生存期模型或软件开发模型，是描述软件过程中各种活动如何执行的模型。它确立了软件开发和演绎各阶段的次序限制以及各阶段活动的准则，确立了开发过程所遵守的规定和限制，便于各种活动的协调以及各种人员的有效通信，有利于活动重

用和活动管理。

目前常见的软件过程模型有瀑布模型、增量模型、螺旋模型、喷泉模型、变换模型和基于知识的模型等。

1.3.1 瀑布模型

瀑布模型是经典的软件开发模型，是 1970 年由 W.Royce 提出的最早的软件开发模型。如图 1-4 所示，瀑布模型将软件开发活动中的各项活动规定为依线性顺序连接的若干个阶段的工作，形如瀑布流水，最终得到软件系统或软件产品。换句话说，它将软件开发过程划分成若干个互相区别而又彼此联系的阶段，每个阶段中的工作都以上一个阶段工作的结果为依据，同时作为下一个阶段的工作基础。

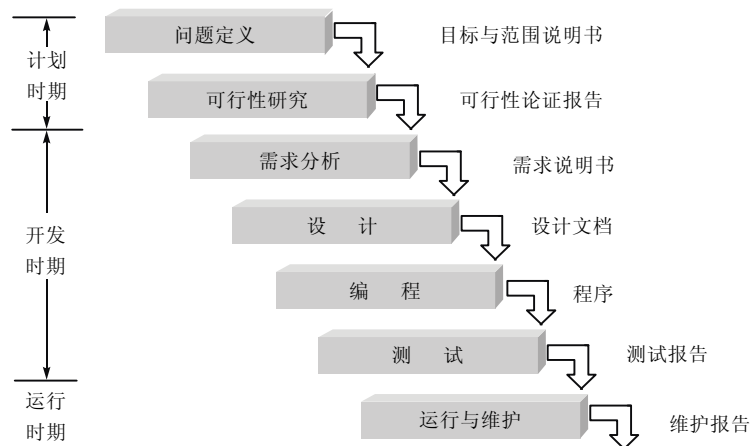


图 1-4 瀑布模型

每个阶段的任务完成之后，产生相应的文档。它是以文档作为驱动的，该模型适合于需求很明确的软件项目开发。

在软件工程的第一阶段，瀑布模型得到了广泛的应用。它简单、易用，在消除非结构化软件、降低软件的复杂性、促进软件开发工程化方面起了很大的作用。但在软件开发实践中也逐渐暴露出它的缺点。由于瀑布模型是一种理想的线性开发模式，它将一个充满回溯的软件开发过程硬性分割为几个阶段，无法解决软件需求不明确或者变动的问题。这些缺点对软件开发带来了严重影响，由于需求不明确，会导致开发的软件不符合用户的需求而夭折。

1.3.2 增量模型

增量模型是一种非整体开发的模型。根据增量的方式和形式的不同，分为基于瀑布模型的渐增模型和基于原型的快速原型模型。图 1-5 描述的一般的增量模型。

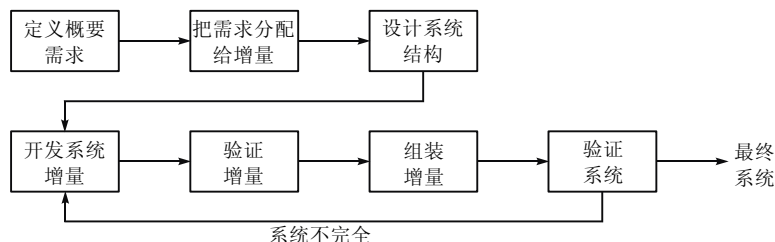


图 1-5 一般增量模型

该模型具有较大的灵活性，适合于软件需求不明确、设计方案有一定风险的软件项目。

增量模型和瀑布模型之间的本质区别是：瀑布模型属于整体开发模型，它规定在开始下一个阶段的工作之前，必须完成前一阶段的所有细节。而增量模型属于非整体开发模型，它推迟某些阶段或所有阶段中的细节，从而较早地产生工作软件。

1.3.3 螺旋模型

对于大型软件，只开发一个原型往往达不到要求。螺旋模型将瀑布模型和增量模型结合起来，并加入了风险分析。它是由 TRW 公司的 B.Boehm 于 1988 年提出的。该模型将开发划分为制定计划、风险分析、实施工程和客户评估 4 类活动。如图 1-6 所示，沿着螺旋线每转一圈，表示开发出一个更完善的新的软件版本。如果开发风险过大，开发机构和客户无法接受，则项目有可能就此终止。在多数情况下，会沿着螺旋线继续下去，自内向外逐步延伸，最终得到满意的软件产品。

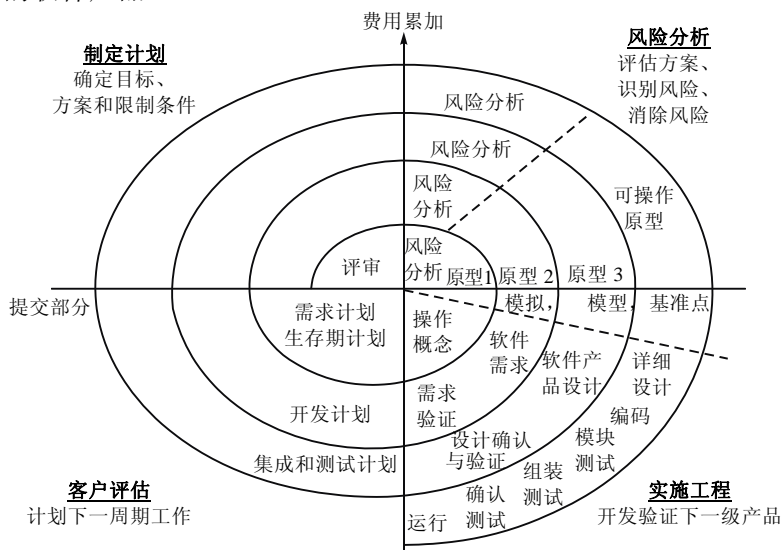


图 1-6 螺旋模型

螺旋模型将开发过程分为几个螺旋周期，每个螺旋周期可分为 4 个工作步骤：

- ① 制定计划：确定目标、方案和限制条件。
- ② 风险分析：评估方案、识别风险和消除风险。

- ③ 实施工程：开发验证下一级产品。
- ④ 客户评估：计划下一周期工作。

1.3.4 喷泉模型

喷泉模型是由 B.H.Sollers 和 J.M.Edwards 于 1990 年提出的一种新的开发模型，主要用于采用面向对象技术的软件开发项目。它克服了瀑布模型不支持软件重用和多项开发活动集成的局限性。喷泉模型使开发过程具有迭代性和无间隙性。软件的某个部分常常被重复使用多次，相关对象在每次迭代中随之加入渐进的软件成分，即迭代性；而分析和设计活动等各项活动之间没有明显的边界，即无间隙性。

喷泉模型是以面向对象的软件开发方法为基础，以用户需求作为喷泉模型的源泉。如图 1-7 所示，喷泉模型有如下特点。

① 喷泉模型规定软件开发过程有 4 个阶段，即分析、系统设计、软件设计和实现。

② 喷泉模型的各阶段相互重叠，它反映了软件过程并行性的特点。

③ 喷泉模型以分析为基础，资源消耗呈塔形，在分析阶段消耗的资源最多。

④ 喷泉模型反映了软件过程迭代性的自然特性，从高层返回低层无资源消耗。

⑤ 喷泉模型强调增量开发，它依据“分析一点，设计一点”的原则，并不要求一个阶段的彻底完成，整个过程是一个迭代的逐步提炼的过程。

⑥ 喷泉模型是对象驱动的过程，对象是所有活动作用的实体，也是项目管理的基本内容。

⑦ 喷泉模型在实现时，由于活动不同，可分为系统实现和对象实现，这既反映了全系统的开发过程，也反映了对象的开发和重用过程。

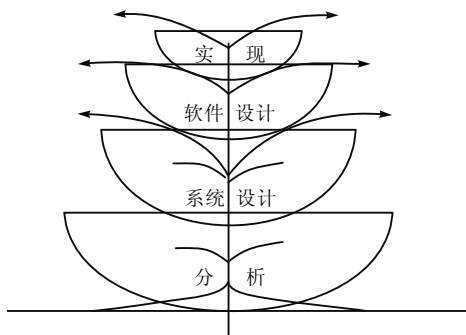


图 1-7 喷泉模型

1.3.5 智能模型

智能模型也称为基于知识的软件开发模型，是知识工程与软件工程在开发模型上结合的产物，是以瀑布模型与专家系统的综合应用为基础建立的模型。该模型通过应用系统的知识和规则帮助设计者认识一个特定的软件的需求和设计，这些专家系统已成为开发过程的伙伴，并指导开发过程。

从图 1-8 中可以清楚地看到，智能模型与其他模型不同，它的维护并不在程序一级上进行，这样大大地降低了问题的复杂性。

智能模型的主要优点有：

- ① 通过领域的专家系统，可使需求说明更加完整、准确和无二义性；
- ② 通过软件工程的专家系统，提供一个设计库支持，在开发过程中成为设计者的助手；



③ 通过软件工程知识和特定应用领域的知识及规则的应用来提供开发的帮助。

但是，要建立适合于软件设计的专家系统，或建立一个既适合软件工程又适合应用领域的知识库都是非常困难的。目前，在软件开发中正应用 AI 技术，在 CASE 工具系统中使用专家系统，例如，使用专家系统来实现测试自动化，AI 技术在软件开发阶段取得了局部进展。

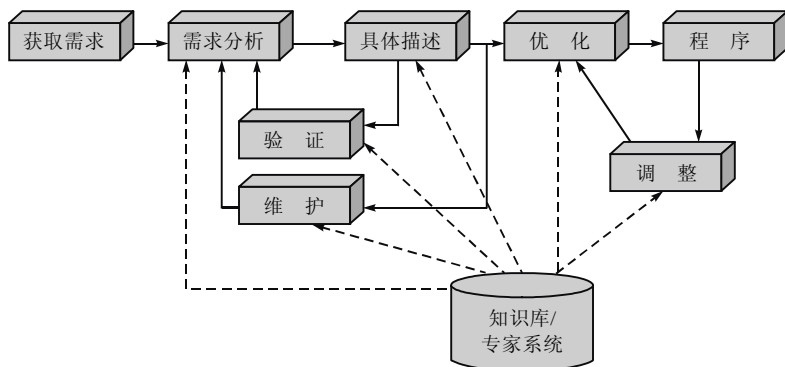


图 1-8 智能模型



1.4 软件开发方法

为了克服软件危机，从 20 世纪 60 年代末开始，各国的软件工作者一直在进行软件开发方法的研究与实践，并取得了一系列研究成果。

软件工程的内容包括技术和管理两方面，两方面又紧密结合。通常，把在软件生命期中所使用的一整套技术的集合称为方法学（Methodology）或范型（Paradigm）。

软件开发方法是一种使用早已定义好的技术集及符号表示习惯来组织软件生产过程的方法，其方法一般表述成一系列的步骤，每一步骤都与相应的技术和符号相关。其目标是在规定的投资和时间内，开发出符合用户要求的。高质量的软件，为此，需要有成功的开发方法。

软件开发方法可分为两大类：面向过程的开发方法和面向对象的开发方法。本节将对结构化开发方法、原型化开发方法和面向对象的开发方法进行简介。



1.4.1 结构化开发方法

结构化开发方法（Structured Developing Method）是一种面向数据流的开发方法，它的基本原则是功能的分解与抽象。结构化方法提出了一组提高软件结构合理性的准则，如分解和抽象、模块的独立性、信息隐蔽等。它是现有的软件开发方法中最成熟、应用最广泛的方法。该方法的主要特点是快速、自然和方便。

结构化方法的指导思想是“自顶向下、逐步求精”。

1. 结构化开发方法的组成

结构化方法由三部分构成，按照推出的先后次序为：

- ◎ 20 世纪 70 年代初推出的结构化程序设计方法 SP (Structured Program) 法;
- ◎ 20 世纪 70 年代中推出的结构化设计方法 SD (Structured Design) 法;
- ◎ 20 世纪 70 年代末推出的结构化分析方法 SA (Structured Analysis) 法。

SA、SD、SP 法相互衔接,形成了一整套开发方法。将 SA、SD 法结合起来,又称为结构化分析与设计技术 (SADT)。

2. 结构化方法的工作模型

结构化方法的工作模型为瀑布模型 (Waterfall Model)。但从 20 世纪 80 年代开始,逐渐发现其不足。软件开发过程是个充满回溯的过程,而瀑布模型却将其分割为独立的几个阶段,不能从本质上反映软件开发过程本身的规律。此外,过分强调复审,并不能完全避免较为频繁的变动。尽管如此,瀑布模型仍然是开发软件产品的一个行之有效的工程模型。

1.4.2 原型化开发方法

原型是指软件开发过程中,软件的一个早期可运行的版本,它反映了最终系统的部分重要特性。原型化方法的基本思想是,花费少量代价建立一个可运行的系统,使用户及早获得学习的机会。原型化开发方法又称速成原型 (Rapid Prototyping) 法,强调的是软件开发人员与用户的不断交互,通过原型的演进不断适应用户任务改变的需求。将维护和修改阶段的工作尽早进行,使用户验收提前,从而使软件产品更加适合实际应用。原型化开发方法有两种方法。

① 快速建立需求规格原型 (Rapid Specification Prototyping, RSP) 法所建立的原型反映了系统的某些特征,让用户学习,有利于获得更加精确的需求说明书,需求说明书一旦确定,原型就被废弃,后阶段的工作仍按照瀑布模型开发,所以也称为废弃 (Throw Away) 型。

② 快速建立渐进原型 (Rapid Cyclic Prototyping, RCP) 法采用循环渐进的开发方式,对系统模型作连续精化,将系统需要具备的功能逐步添加上去,直至所有功能全部满足,此时的原型模型也就是最终的产品,所以也称为追加 (add on) 型。

原型化开发方法适合于开发探索型、实验型与进化型的软件系统。其工作模型如图 1-9 所示,这是一个循环的模型。原型化开发方法按以下步骤循环执行。

① 快速分析。快速确定软件系统的基本要求,确定原型所要体现的特征 (界面、总体结构、功能、性能)。

② 构造原型。在快速分析的基础上,根据基本规格说明,忽略细节,只考虑主要特征,快速构造一个可运行的系统。有三类原型:用户界面原型、功能原型和性能原型。

③ 运行和评价原型。用户试用原型并与开发者之间频繁交流,目的是验证原型的正确性。

④ 修改与改进。对原型进行修改、增删。

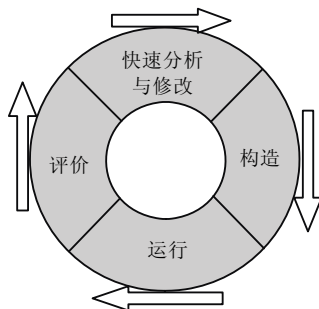


图 1-9 原型化开发方法工作模型



1.4.3 面向对象的软件开发方法

面向对象的软件开发方法（Object-Oriented Software Development, OOSD）是 20 世纪 80 年代推出的一种全新的软件开发方法。OOSD 非常实用且强有力，被誉为 20 世纪 90 年代软件的核心技术之一。

其基本思想是：对问题领域进行自然的分割，以更接近人类通常思维的方式建立问题领域的模型，以便对客观的信息实体进行结构和行为的模拟，从而使设计的软件更直接地表现问题的求解过程。面向对象的开发方法以对象作为最基本的元素，是分析和解决问题的核心。

Coad 和 Yourdon 给出一个面向对象的定义如下：

面向对象=对象+类+继承+消息

如果一个软件系统是按照这样 4 个概念设计和实现的，则可以认为这个软件系统是面向对象的。

OOSD 由 OOA（面向对象的分析）、OOD（面向对象的设计）和 OOP（面向对象的程序设计）三部分组成。

（1）OOA（Object-Oriented Analysis）法

OOA 就是解决“做什么”的问题。它的基本任务就是建立以下三种模型。

对象模型（信息模型）——定义构成系统的类和对象，以及它们的属性与操作。

状态模型（动态模型）——描述任何时刻对象的联系及其联系的变化，即时序。常用状态图与事件追踪图描述。

处理模型（函数模型）——描述系统内部数据的传送处理。

显然，在三大模型中，最重要的是对象模型。如何建立三大模型，我们将在第 5 章中介绍。

（2）OOD（Object-Oriented Design）法

在需求分析的基础上，进一步解决“如何做”的问题。OOD 法分为概要设计和详细设计两步。

概要设计：细化对象行为，添加新对象，认定类，组类库，确定外部接口及主要数据结构。

详细设计：详细的对象描述。

（3）OOP（Object-Oriented Program）法

使用面向对象的程序设计语言，如 C++ 进行程序设计，因为该类语言支持对象、类、多态性和继承等概念，因此比较容易实现面向对象的程序设计。

用面向对象方法开发的软件，其结构基于客观世界界定的对象结构，因此与传统的软件相比较，软件本身的内容结构发生了质的变化，因而易复用性和易扩充性都得到了提高，而且能支持需求的变化。



1.5 软件工具与软件开发环境

为支持软件开发、维护、管理而研制的计算机程序系统称为软件工具。像程序系统可分为系统和子系統一样，软件开发工具也可具有不同的粒度，称之为工具或工具片断。软件工具通常由工具、工具接口和工具用户接口三部分构成。工具通过工具接口与其他工具、操作系统或网络操作系统及通信接口、环境信息库接口等进行交互作用，当工具需要与用户进行交互作用时，则通过工具的用户接口来进行。

软件工具种类繁多，涉及面广，如编辑、编译、正文格式处理、静态分析、动态跟踪、需求分析、设计分析、测试、模拟和图形交互等。

在软件工程活动中，软件工程师和管理员按照软件工程的方法和原则，借助于计算机及其软件工具的帮助，开发、维护、管理软件产品的过程，称为计算机辅助软件工程（Computer-Aided Software Engineering, CASE）。CASE 的实质是为软件开发提供一组优化集成的且节省大量人力的软件开发工具，其目的是实现软件生存周期各环节的自动化并使之成为一个整体。

CASE 使用多种软件工具，这些软件工具分为两个层次。

（1）依赖于软件内生命周期各阶段的分散工具

这些工具只能支持软件开发某个阶段的工具，而不能支持整个软件生命周期。例如，美国密执安大学 ISDOS 项目组研制的 PSL/PSA 系统，为分析员提供编写和检查需求分析文档的工具。PSL（Problem Statement Language）是问题说明语言，它可以按照一定的语法描述用户对系统的功能和性能要求。PSA（Problem Statement Analyzer）是问题说明分析器，可以对用 PSA 写的文本进行分析，产生许多有用的报告。

又如，美国 Tektronix 公司针对 SA 法开发的 Tektronix 工具箱，可对 SA 文档中的图形、文字进行编辑和查错，还可从 SA 文档中自动导出初始的模块结构图。

美国 Hughes 飞机公司开发的概要设计工具 AIDES（Automated Interactive Design and Evaluation System），也只提供对 SD 阶段的模块图的绘制和文档管理功能。

（2）软件开发环境（Software Development Environment）

软件开发工具也称为软件工程环境（Software Engineering Environment），是包括方法、工具和管理等多种技术在内的综合系统。好的软件开发环境能够简化软件开发过程，提高软件开发质量。它应具备以下特点：

- ◎ 紧密性，各种工具紧密配合工作；
- ◎ 坚定性，环境可自我保护，不受用户和系统影响，可实现非预见性的环境恢复；
- ◎ 可适应性，适应用户要求，环境中的工具可修改、增加或减少；
- ◎ 可移植性，指工具可移植。

如图 1-10 所示，典型的软件工程环境具有三级结构：

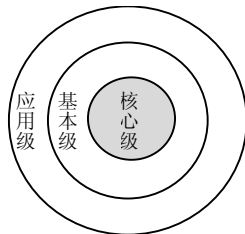


图 1-10 典型的软件工程环境



- ◎ **核心级**——包括核心工具组、数据库、通信工具、运行支持、功能、与硬件无关的移植接口等；
- ◎ **基本级**——包括环境的用户工具，编译、编辑程序，作业控制语言的解释程序等；
- ◎ **应用级**——通常指应用软件的开发工具。

习题 1

- 1.1 软件产品的特性是什么？
- 1.2 软件发展分为几个阶段？各有何特征？
- 1.3 什么是软件危机？其产生的原因是什么？
- 1.4 什么是软件过程？有哪些主要的软件过程模型？它们各有哪些特点？
- 1.5 主要的软件开发方法有哪些？
- 1.6 软件生命期各阶段的主要任务是什么？
- 1.7 原型化开发方法的核心是什么？它具有哪些特点？

第2章 软件维护



2.1 软件维护的基本概念

软件开发完成交付用户使用后，就进入软件的运行和维护阶段。软件维护是指软件系统交付使用以后，为了改正软件运行错误，或者为满足新的需求而加入新功能的修改软件的过程。

软件维护工作处于软件生存期的最后阶段，维护阶段是软件生存期中最长的一个阶段，所花费的人力、物力最多，其花费高达整个软件生存期花费的 60%~70%。因为计算机程序总是会发生变化，包括对隐含错误的修改，新功能的加入，环境变化造成的程序变动等。因此，应该充分认识到维护工作的重要性和迫切性，提高软件的可维护性，减少维护的工作量和费用，延长已经开发软件的生命期，以发挥其应有的效益。



2.1.1 软件维护的目的

软件维护是软件工程的一个重要任务，其主要工作就是在软件运行和维护阶段对软件产品进行必要的调整和修改。一般来说，要求进行维护的原因主要有以下 5 种。

- ① 在运行中发现在测试阶段未能发现的潜在的软件错误和设计缺陷。
- ② 根据实际情况，需要改进软件设计，以增强软件的功能，提高软件的性能。
- ③ 要求在某环境下已运行的软件能适应特定的硬件、软件、外部设备和通信设备等新的工作环境，或要求适应已变动的数据或文件。
- ④ 为使投入运行的软件与其他相关的程序有良好的接口，以利于协同工作。
- ⑤ 为使运行软件的应用范围得到必要的扩充。

随着计算机功能越来越强，社会对计算机的需求越来越大，要求软件必须快速发展。在软件快速发展的同时，应该考虑软件的开发成本，显然，对软件进行维护的目的是为了纠正软件开发过程未发现的错误，增强、改进和完善软件的功能和性能，以适应软件的发展，延长软件的寿命，让其创造更多的价值。



2.1.2 软件维护的类型

根据前面提到的软件维护的目的可以把维护活动归纳为完善性维护、适应性维护、纠错性维护和预防性维护 4 类。各类维护比例如图 2-1 所示。



1. 完善性维护（Perfective Maintenance）

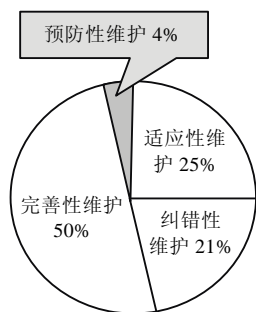


图 2-1 各类维护所占的比例

在软件漫长的使用过程中，为了满足用户使用过程中对软件提出的新的功能与性能要求，需要对原来的软件功能进行修改或扩充，这种扩充软件功能、增强软件性能、提高软件运行效率和可维护性而进行的维护活动称为完善性维护。此类维护活动工作量较大，约占整个维护工作的 50%。例如，对人事管理程序，在使用中可能要增加或删除新的项目，从满足新需求；原来软件的查询响应速度较慢，要提高软件的响应速度；改变原来软件的用户界面或增加联机帮助信息；为软件的运行增加监控设施等。

主要维护策略是，使用功能强大、使用方便的工具，采用原型化开发方法等。

2. 适应性维护（Adaptive Maintenance）

适应性维护是为了适应计算机技术的飞速发展，使软件适应外部新的硬件和软件环境或者数据环境（数据库、数据格式、数据输入/输出方式、数据存储介质）发生的变化，而进行修改软件的过程。适应性维护约占整个维护工作的 25%。例如，为现有的某个应用问题实现一个数据库管理系统；对某个指定代码进行修改，如：从 3 个字符改为 4 个字符；缩短系统的应答时间，使其达到特定的要求；修改两个程序，使它们可以使用相同的记录结构；修改程序，使其适用于另外的终端。

主要的维护策略是，对可能变化的因素进行配置管理，将因环境变化而必须修改的部分局部化，即局限于某些程序模块等。

3. 纠错性维护（Corrective Maintenance）

软件测试不可能找出一个软件系统中所有潜伏的错误，所以当软件在特定情况下运行时，这些潜伏的错误可能会暴露出来。对在测试阶段未能发现的，在软件投入使用后才逐渐暴露出来的错误进行测试、诊断、定位、纠错及验证、修改的回归测试过程，称为纠错性维护。纠错性维护占整个维护工作的 21%，例如，修正原来程序中并未使开关复位的错误；解决开发时未能测试各种可能条件带来的问题；解决原来程序中遗漏处理文件最后一个记录的问题。

主要维护策略是，在开发过程中采用新技术，利用应用软件包，提高系统结构化程度，进行周期性维护审查等。

4. 预防性维护（Preventive Maintenance）

预防性维护是为了提高软件的可维护性和可靠性，采用先进的软件工程方法对需要维护的软件或软件中的某一部分重新进行设计、编制和测试，为以后进一步维护和运行打好基础。也就是说，软件开发组织选择在最近的将来可能变更的程序，做好变更它们的准备。对该类维护工作必须采用先进的软件工程方法，对需要修改的软件或部分进行设计、编码和测试。对该类维护工作的必要性有争议，所以它在整个维护活动中占较小的比例，约占 4%。例如，

预先选定多年留待使用的程序；当前正在成功使用着的程序；可能在最近的将来要做重大修改或增强的程序。

主要维护策略是，采用提前实现、软件重用等技术。

2.1.3 软件维护的特性

1. 时间长、工作量大、成本高

软件的维护过程是软件生存期中最长的，并且相当困难的阶段。软件维护的工作量占整个软件生存期的 70% 以上，而且还在逐年增加。因此，如何减少软件维护的工作量，降低软件维护的成本，就成为提高软件维护效率和质量的关键。

2. 维护的副作用

通过维护可以延长软件的寿命，使其创造更多的价值，但是，修改软件是危险的，每修改一次，都可能会产生新的潜在错误，因此，维护的副作用是指由于修改程序而导致新的错误或者新增加一些不必要的活动。一般维护产生的副作用有如下三种。

① 修改代码的副作用。在修改源代码时，由于软件的内在结构等原因，任何一个小的修改都可能引起错误。因此在修改代码时必须特别小心。

② 修改数据的副作用。在修改数据结构时，有可能造成软件设计与数据结构的不匹配，因而导致软件出错。数据副作用就是修改软件信息结构导致的结果。修改数据副作用可以通过详细的设计文档加以控制，此文档中描述了一种交叉作用，把数据元素、记录、文件和其他结构联系起来。

③ 修改文档的副作用。对软件的数据流、软件结构、模块逻辑等进行修改时，必须对相关技术文档进行相应的修改。但是修改文档过程会产生新的错误，导致文档与程序功能不匹配，默认条件改变等错误，产生修改文档的副作用。

为了控制因修改而引起的副作用，应该按模块把修改分组；自顶向下地安排被修改模块的顺序；每次修改一个模块。

3. 软件维护的困难

由于软件维护工作通常并不是由软件的设计和开发人员来完成的，因此维护人员首先要对软件各阶段的文档和代码进行分析、理解。因而出现了理解程序困难、文档不齐等问题，尤其是对大型、复杂系统的维护，更加困难和复杂。

(1) 结构化维护和非结构化维护

① 非结构化维护是指只有源程序，缺乏必要的文档说明，难于确定数据结构、系统接口等特性，维护工作令人生畏。

② 结构化维护是指软件开发过程按照软件工程方法进行，开发各阶段文档齐全，软件的维护有完整的方案、技术、审定过程。

可以看到，维护工作的难度及工作量的大小，明显与前期的开发工作密切相关。

(2) 维护的困难

软件维护的困难主要是由于软件需求分析和开发方法的缺陷造成的，这些困难主要



包括：

- ① 读懂别人编的程序困难；
- ② 文档的不一致性；
- ③ 软件开发人员和软件维护人员在时间上有差异；
- ④ 软件维护工作难出成果，大家都不愿意干。



2.1.4 软件维护的代价

1. 软件维护的工作量大

软件维护的费用是整个软件开发费用的 55%~70%，并且其所占比例在逐年上升。而且，维护中还可能产生新的潜在错误。例如，1970 年维护费用约占软件开发费用的 40%，到 1990 年，维护费用所占比例就超过了 70%。另外，维护还包含了无形的资源占用，包括大量使用硬件、软件和软件工程师等资源。

在软件维护时，直接影响维护成本和工作量的因素很多。

(1) 系统规模大小

系统规模大小直接影响维护工作量。系统规模越大，仅仅看懂程序就很困难，维护的工作量就更大。系统规模主要由源代码行数、程序模块数、数据接口文件数、使用数据库规模大小等因素决定。

(2) 程序设计语言

参与软件开发的人员都知道，解决相同的问题，如果选择不同的程序设计语言，得到的程序的规模可能不同，由此应选择功能强且适合解决问题的程序设计语言，这样可以使生成程序的指令数更少。

(3) 系统使用年限

使用年限长的老系统维护比新系统所需要的工作量更大。老系统因已经进行多次维护，参与维护的人员也不断变化，因此这样的系统结构更乱，如果没有系统说明和设计文档，维护就更加困难。

(4) 软件开发新技术的应用

在开发过程中，尽量使用先进的分析和设计技术及程序设计技术，如面向对象的技术、构件技术、可视化程序设计技术等，可以减少维护工作量。

(5) 设计过程中的技术

在具体对软件进行维护时，影响维护工作量的其他因素还有很多，例如，设计过程中应用的类型、数学模型、任务的难度、开关与标记、IF 嵌套深度、索引或下标数等。

2. 软件维护工作量模型


维护活动分为生产性活动和非生产性活动两类。生产性活动包括分析评价、修改设计和编写程序代码等。非生产性活动包括理解程序代码，解释数据结构，接口特点和设计约束等。

Belady 和 Lehman 提出了软件维护工作模型：

$$M=P+K\times\text{EXP}\left(C-D\right)$$

式中， M 表示维护总工作量， P 表示生产性活动工作量， K 为经验常数， C 表示由非结构化维护引起的程序复杂度， D 表示对维护软件熟悉程度的度量。

从上式可见， C 越大， D 越小，则维护工作量呈指数的增加。 C 增大主要因为软件采用非结构化设计，程序复杂性高； D 减小则表示维护人员不是原来的开发人员，不熟悉程序，理解程序需要花费较长时间。



2.2 软件维护的过程

软件维护是一件复杂而困难的事，必须在相应的技术指导下，按照一定的步骤进行。首先要建立一个维护的组织，建立维护活动的登记、申请制度，以及对维护方案的审批制度，规定复审的评价标准。

通过软件维护组织对维护过程进行有效的控制，例如，首先要对软件进行全面、准确、迅速的理解，这是决定维护工作成败和质量好坏的关键。

1. 维护组织

除了较大的软件公司外，通常在软件维护工作方面，并不保持一个正式的组织。在软件开发部门，确立一个非正式的维护组织即非正式的维护管理员来负责维护工作却是绝对必要的。图 2-2 给出了一种典型的软件维护组织方式。

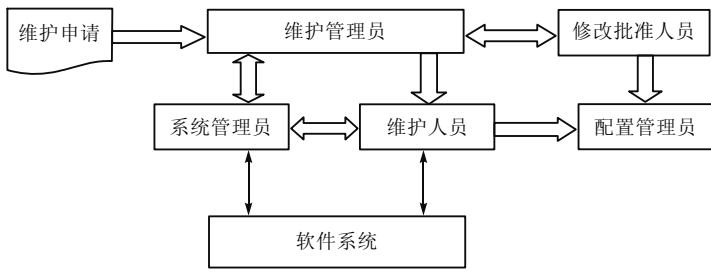


图 2-2 软件维护组织方式

维护管理员可以是某个人，也可以是一个包括管理人员、高级技术人员等在内的小组。维护管理员将提交的维护申请交给系统管理员进行评价，然后由修改批准人员决定如何修改，交给维护人员在系统管理人员的指导下对软件进行修改。在修改过程中，配置管理员对软件配置进行审查。

2. 维护工作的流程

图 2-3 描述了实施软件维护的工作流程，根据用户或维护人员的更改要求，在维护申请经过评审后，首先要确定维护的类型，还要分辨错误的严重程度或修改优先级的高低，分别处理。

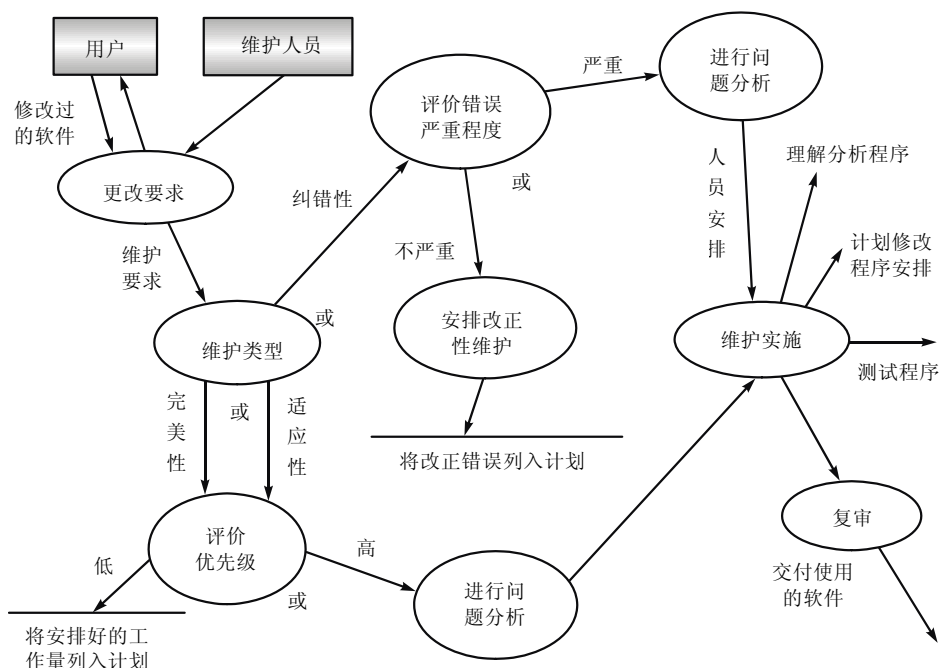


图 2-3 软件维护的工作流程图

3. 维护工作的组织管理

软件维护工作不仅是技术性的，还需要大量的管理工作与之相配合，才能保证维护工作的质量。管理部门应对提交的修改方案进行分析和审查，并对修改带来的影响进行充分的估计，对于不恰当的修改予以撤销。需要修改主文档时，管理部门更应仔细审查。

软件维护的管理流程如图 2-4 所示。

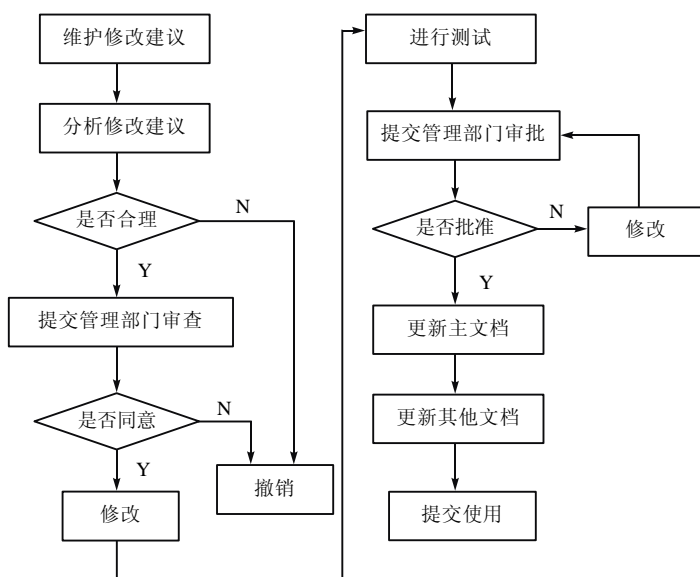


图 2-4 软件维护管理流程图



2.3 软件维护技术

正确、合理地使用软件维护技术,是提高维护效率和质量的关键。软件维护技术包括:

- ◎ 面向维护的技术——涉及软件开发的所有阶段;
- ◎ 维护支援技术——支持软件维护阶段的技术;
- ◎ 维护档案记录——做好维护档案记录,才能为维护评价提供有效的数据;
- ◎ 维护评价——确定维护的质量和成本。

1. 面向维护的技术

面向维护的技术是指软件开发阶段用来减少错误、提高软件可维护性的技术。它涉及软件开发的所有阶段。

需求分析阶段:对用户的需求进行严格的分析定义,使之没有矛盾且易于理解,可以减少软件中的错误。例如,美国密执安大学的 ISDOS 系统就是在需求分析阶段使用的一种分析和文档化工具,可以用它来检查需求说明书的一致性和完备性。

设计阶段:划分模块时充分考虑将来改动或扩充的可能性,采用结构化分析和结构化设计方法,以及通用的硬件和操作系统来进行设计。

编码阶段:使用灵活的数据结构,使程序相对独立于数据的物理结构,养成良好的程序设计习惯。

测试阶段:尽可能多发现错误,保存测试所用例子及测试数据等。

这些技术都可以减少软件错误,提高软件的可维护性。

2. 维护支援技术

维护支援技术是指在软件维护阶段用来提高维护作业的效率和质量的技术,包括以下内容。

- ① 信息收集:收集有关系统在运行过程中的各种问题。
- ② 错误原因分析:分析所收集到的信息,找到出错的原因。
- ③ 软件分析与理解:只有对需要维护的软件进行认真的理解,才能保证软件维护正确进行。
- ④ 维护方案评价:在进行维护修改前,要确定维护方案,并由相关的组织进行评审通过后才能执行。
- ⑤ 代码与文档修改:实施维护方案。
- ⑥ 修改后的确认:经过修改的软件,需要重新进行测试。
- ⑦ 远距离的维护:对于网络系统,可以通过远程控制进行维护。

3. 维护档案记录

为了更好地做好软件维护工作,包括估计维护的有效程度,确定软件产品的质量,确定维护的实际开销等,应该在维护的过程中记录好维护全过程,建立维护档案。

维护档案记录的内容应该全面详细地记录相关信息。Swanson 提出维护档案记录的内容包括:程序名称,源程序语句条数,机器代码指令条数,所用的程序设计语言,程序安装的



日期, 程序安装后的运行次数, 程序安装后运行次数有关的处理故障数, 程序改变的层次及名称, 修改程序所增加的源程序语句数, 修改程序所减少的源程序语句数, 每次修改所付出的“人时”数, 修改程序的日期, 软件维护人员的姓名, 维护申请报告的名称, 维护类型, 维护开始时间和结束时间, 花费在维护上的累计“人时”数, 维护工作的净收益等。

对每项维护任务, 都应该详细记录这些数据。

4. 维护评价

对于维护工作的评价一般较为困难, 因为很多时候没有量化的数据, 但可以记录维护性能的度量值, 这些度量值一般包括:

- ◎ 记录维护申请报告的平均处理时间;
- ◎ 统计每类维护的总“人时”数的开销;
- ◎ 统计每次程序运行时的平均出错次数;
- ◎ 统计在维护中, 增删每条源程序语句所花费的平均“人时”数;
- ◎ 记录每段程序、每种语言、每种维护类型的程序平均修改次数;
- ◎ 统计所用语言及用于每种语言的平均“人时”数;
- ◎ 计算各类维护申请的百分比。
- ◎ 这些度量值提供的是定量数据, 可以评价维护工作。



2.4 软件可维护性

许多软件的维护很困难, 主要是因为软件的源程序和文档难于理解和修改。由于维护工作面广, 维护的难度大, 稍有不慎, 就会在修改时给软件带来新的问题或引入新的错误, 所以为了使软件易于维护, 必须考虑软件的可维护性。

2.4.1 软件可维护性的定义

软件可维护性是指软件能够被理解, 并能纠正软件系统出现的错误和缺陷, 以及为满足新的要求进行修改、扩充或压缩的容易程度。软件的可维护性、可使用性和可靠性是衡量软件质量的几个主要特性, 也是用户最关心的问题之一。但影响软件质量的这些因素, 目前还没有普遍适用的定量度量的方法。

软件的可维护性是软件开发阶段各个时期的关键目标。影响软件可维护性的因素很多, 设计、编码和测试中的疏忽, 低劣的软件配置, 缺少文档等都对软件的可维护性产生不良影响。软件维护可用如下的 7 个质量特性来衡量, 即可理解性、可测试性、可修改性、可靠性、可移植性、可使用性和效率。而且对于不同类型的维护, 这 7 个特性的侧重点也不相同。这些质量特性通常体现在软件产品的许多方面。将这些特性作为基本要求, 需要在软件开发的整个阶段采用相应的措施保证, 也就是说, 这些质量要求渗透到软件开发的各个步骤中。因此, 软件的可维护性是产品按照这 7 个质量特性要求进行开发的最终结果。目前广泛使用的衡量程序的可维护性的 7 个特性如表 2-1 所示。

表 2-1 衡量软件可维护性的 7 个质量特性

	改正性维护	适应性维护	完善性维护
可理解性	√		
可测试性	√		
可修改性	√	√	
可靠性	√		
可移植性		√	
可使用性		√	√
效率			√

由于许多质量特性是相互抵触的，要考虑几种不同的度量标准，去度量不同的质量特性。

1. 可理解性

可理解性表明人们通过阅读源代码和相关文档，了解软件功能和运行状况的容易程度。一个可理解的软件主要应该具备的特性是：模块化、风格一致性、使用清晰明确的代码、使用有意义的数 据名和过程名、结构化、完整性等。

对于可理解性，可以用“90-10 测试法”来进行衡量，即让有经验的程序员先用 10 分钟的时间阅读要测试的程序，然后如果能凭记忆和理解写出 90%的程序，则称该程序是可理解的。

2. 可靠性

可靠性表明一个软件按照用户的要求和设计目标，在给定的一段时间内正确执行的概率。可靠性的主要度量标准有：平均失效间隔时间、平均修复时间、有效性。度量可靠性的方法，主要有两种：

- ① 根据软件错误统计数字，进行可靠性预测；
- ② 根据软件复杂性，预测软件可靠性。

3. 可测试性

可测试性表明论证软件正确性的容易程度。对于软件中的程序模块，可用程序复杂性来度量可测试性。明显地，程序的环路复杂性越大，程序的路径就越多，全面测试程序的难度就越大。

4. 可修改性

可修改性表明软件容易修改的程度。一个可修改的软件应当是：可理解的、通用的、灵活的、简单的。其中，通用是指软件适用的功能发生改变而无须修改。灵活是指对软件进行修改很容易。

测试可修改性的一种定量方法是修改练习。其基本思想是，通过做几个简单的修改，来评价修改难度。设 C 是程序中各个模块的平均复杂性， n 是必须修改的模块数， A 是要修改的模块的平均复杂性。则修改的难度表示为：

$$D=A/C$$



在简单修改时,若 $D > 1$,则说明该软件修改困难。 A 和 C 可用任何一种度量程序复杂性的方法计算。

5. 可移植性

可移植性表明软件转移到一个新的计算环境的可能性的_{大小},或者软件能有效地在各种环境中运行的容易程度。一个可移植性好的软件应具有良好、灵活、不依赖于某一具体计算机或操作系统的性能。

6. 效率

效率表明一个软件能执行预定功能而又不浪费机器资源的程度,包括:内存容量、外存容量、通道容量和执行时间。

7. 可使用性

从用户的角度出发,可使用性是软件方便、实用及易于使用的程度。一个可使用的程序应该易于使用,允许出错和进行修改,而且尽量保证用户在使用时不陷入混乱状态。

2.4.2 提高可维护性的方法

软件的可维护性对于延长软件的生存期具有决定意义,因此必须考虑怎样才能提高软件的可维护性。为此,可从以下 5 个方面着手。

1. 建立明确的软件质量目标

要让程序完全满足可维护性的 7 种质量特性,肯定是很难实现的。实际上,某些质量特性是相互促进的,如可理解性与可测试性,可理解性与可修改性;而某些质量特性是相互抵触的,如效率与可移植性,效率与可修改性。因此,为保证程序的可维护性,应该在一定程度上满足可维护的各个特性,但各个特性的重要性又是随着程序的用途或计算机环境的不同而改变的。对编译程序来说,效率和可移植性可能是主要的;对信息管理系统来说,可使用性和可修改性可能是主要的。通过实验证明,强调效率的程序包含的错误比强调简明性的程序所包含的错误要高出 10 倍。所以,在提出目标的同时还必须规定它们的优先级,这样有助于提高软件的质量。

2. 使用先进的软件开发技术和工具

利用先进的软件开发技术是软件开发过程中提高软件质量,降低成本的有效方法之一,也是提高可维护性的有效方法。常用的技术有:模块化、结构化程序设计,自动重建结构和重新格式化等。

例如,面向对象的软件开发方法就是一种非常实用而强有力的软件开发方法。面向对象方法按照人的思维方法,用现实世界的概念来思考问题,这样能自然地解决问题。它强调模拟现实世界中的概念而不是强调算法,鼓励开发者在开发过程中按应用领域的实际概念思考建立模型,模拟客观世界,使描述问题的_{问题空间}和解空间尽量一致,开发出尽量直观、自然的表现求解方法的软件系统。

面向对象方法开发的软件系统具有较好的稳定性。传统方法开发出来的软件系统的结构紧密程度依赖于系统所具有的功能。当功能发生变化时，会引起软件结构的整体修改，因此这样的软件结构是不稳定的。面向对象方法以对象为中心构造软件系统，用对象模拟问题中的实体，以对象间的联系表示实体间的联系，根据问题领域中的模型来建立软件系统的结构。由于客观世界的实体之间的联系是相对稳定的，因此建立的模型也相对稳定。当系统功能需求发生变化时，不会引起软件结构的整体变化，往往只需做一些局部修改。

采用面向对象方法构造的软件可重用性好。对象所固有的封装性和信息隐蔽机制，使得对象内部的实现和外界隔离，具有较强的独立性。因此对象类提供了较为理想的模块化机制，其可重用性自然很好。

采用面向对象方法构造的软件模块的独立性好，修改一个类很少会影响其他类。如果类的接口不变，则只需修改内部，而软件的其他部分都不会受到影响。同时，面向对象的软件符合人们习惯的思维方式，用此方法构造的软件结构与问题空间的结构基本一致，因此面向对象的软件系统比较容易理解。

对面向对象的软件进行维护，主要通过从已经有的类派生出一些新类的维护来实现。因此，维护时的测试和调试工作也主要围绕这些新派生出来的类进行。对类的测试通常比较容易实现，如果发现错误也往往在类的内部，比较容易测试。

总之，面向对象方法开发出来的软件系统，稳定性好、容易修改、易于测试和调试，因此可维护性好。

3. 建立明确的质量保证工作

质量保证是指为提高软件质量所做的各种检查工作。在软件开发和软件维护的各个阶段，质量保证检查是非常有效的方法。为了保证软件的可维护性，可以进行 4 种类型的软件检查。

(1) 在检查点进行复审

检查点是软件开发过程每一个阶段的终点。检查点进行检查的目标是证实已开发的软件是满足设计要求的。保证软件质量的最佳方法是在软件开发的最初阶段就把质量要求考虑进去，并在每个阶段的终点设置检查点进行检查，如图 2-5 所示。在不同的检查点，检查的重点不完全相同，各阶段的检查重点、对象和方法参见表 2-2。

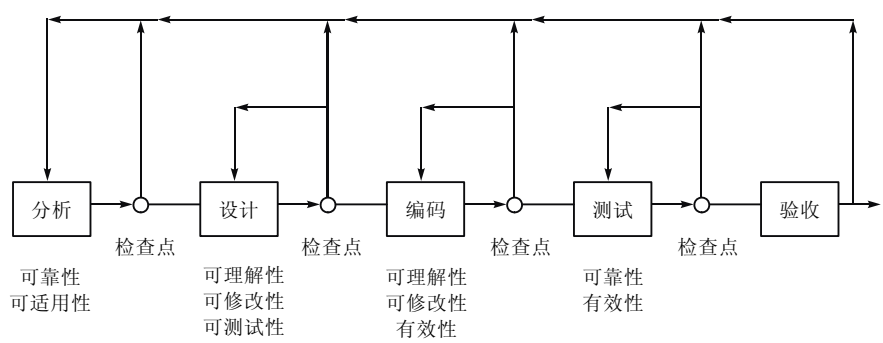




表 2-2 各阶段的检查重点、对象和方法

	检 查 重 点	检 查 项 目	检查方法或工具
需求分析	对程序可维护性的要求是什么？例如，可使用性，交互系统的响应时间	1. 软件需求说明书 2. 限制与条件，优先顺序 3. 进度计划 4. 测试计划	可使用性检查表
设计	1. 程序是否可理解 2. 程序是否可修改 3. 程序是否可测试	1. 设计方法 2. 设计内容 3. 进度 4. 运行、维护支持计划	1. 复杂性度量、标准 2. 修改练习 3. 耦合、内聚估算 4. 可测试性检查表
编码及单元测试	1. 程序是否可理解 2. 程序是否可修改 3. 程序是否可移植 4. 程序是否效率高	1. 源程序清单 2. 文档 3. 程序复杂性 4. 单元测试结果	1. 复杂性度量、90-10 测试、自动结构检查程序 2. 可修改性检查表、修改练习 3. 编译结果分析 4. 效率检查表、编译对时间和空间的要求
组装与测试	1. 程序是否可靠 2. 程序是否效率高 3. 程序是否可移植 4. 程序是否可使用	1. 测试结果 2. 用户文档 3. 程序和数据文档 4. 操作文档	1. 调试、错误统计、可靠性模型 2. 效率检查表 3. 比较在不同计算机上的运行结果 4. 验收测试结果、可使用性检查表

(2) 验收检查

验收检查是一个特殊的检查点的检查，它是把软件从开发转移到维护的最后一次检查。它对减少维护费用，提高软件质量非常重要。

① 需求和规范标准。以需求规格说明书为标准，进行检查，区分必须的、任选的、将来的需求，包括系统运行时对计算机设备的需求，对维护、测试、操作，以及维护人员的需求，对测试工具的需求等。

② 设计标准。软件应设计成分层的模块结构。每个模块应完成独立的功能，满足高内聚、低耦合的原则。通过一些知道预期变化的实例，说明设计的可扩充性、可缩减性和可适应性。

③ 源代码标准。所有的代码都必须具有良好的结构，所用的代码都必须文档化，在注释中说明它的输入、输出，以及便于测试/再测试的一些特点与风格。

④ 文档标准。文档中应说明程序的输入/输出、使用方法/算法、错误恢复方法、所有参数的范围及默认条件等。

(3) 周期性的维护检查

上述两种软件检查可用来保证新的软件系统的可维护性。对已运行的软件应该进行周期性的维护检查。为了纠正在开发阶段未发现的错误和缺陷，使软件适应新的计算机环境并满足变化的用户要求，对正在使用的软件进行改变是不可避免的。改变程序可能引起新的错误并破坏原来程序概念的完整性。为了保证软件质量，应该对正在使用的软件进行周期性维护检查。实际上，周期性维护检查是开发阶段对检查点进行检查的继续，采用的检查方法和内

容都是相同的。把多次检查的结果与以前进行的验收检查结果和检查点检查结果进行比较，对检查结果的任何变化进行分析，并找出原因。

(4) 对软件包进行检查

上述三种方法适用于组织内部开发和维护的软件或专为少量用户设计的软件，很难适用于有很多用户的通用软件包。因软件包属于卖方的资产，用户很难获得软件包源代码和完整的文档。对这种软件包的维护方法是，单位的维护程序员在分析研究卖方提供的用户手册、操作手册、培训手册、新版本策略指导、计算机环境和验收测试的基础上，深入了解本单位的希望和要求，编制软件包检验程序。软件包检测程序是一个测试程序，它检查软件包程序所执行的功能是否与用户的要求和条件相一致。

4. 选择可维护的程序设计语言

程序设计语言的选择对维护影响很大。很明显，低级语言很难理解和掌握，维护当然很困难。高级语言比低级语言更容易理解，在各种高级语言中，一些语言可能比另一些语言更容易理解。

第四代语言，例如查询语言、图形语言、报表生成语言和非常高级语言等，对减少维护费用来说是最有吸引力的语言。人们容易理解、使用和修改它们。例如，用户使用第四代语言开发商业应用程序比使用通常的高级语言快很多倍。

程序设计语言对软件可维护性的影响如图 2-6 所示。

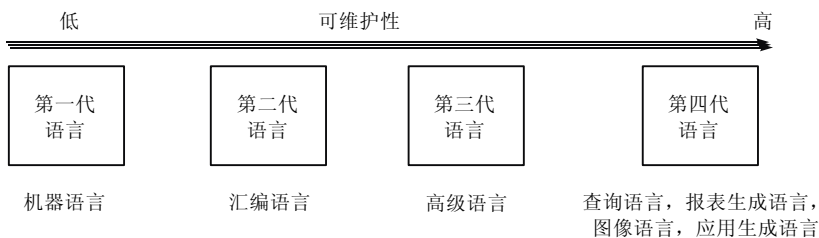


图 2-6 程序设计语言对软件可维护性的影响

一些第四代语言是过程语言，而另一些是非过程语言。对非过程语言，用户不需要指出实现算法，只需向编译程序或解释程序提出自己的要求即可。例如，它能自动选择报表格式、文字字符类型等。自动生成指令能改进软件的可靠性。另外，第四代语言容易理解，容易编程，程序容易修改，因此改进了可维护性。

5. 改进程序的文档

(1) 程序文档

程序员利用程序文档来解释和理解程序的内部结构，以及程序同系统内其他程序、操作系统和其他软件系统如何相互作用。程序文档包括源代码注释、设计文档、系统流程、程序流程图和交叉引用表等。

程序文档是对程序的总目标、程序的各组成部分之间的关系、程序设计策略、程序时间过程的历史数据等的说明和补充。程序文档能提高程序的可阅读性。为了维护程序，人们不得不阅读和理解程序文档。虽然人们对程序的看法不一，但普遍同意以下观点：



- ① 好的文档能使程序更容易阅读，坏的文档比没有更糟；
- ② 好的文档简明扼要，风格统一，容易修改；
- ③ 程序编码中加入必要的注释可提高程序的可理解性；
- ④ 程序越长、越复杂，越应该注重程序文档的编写。

(2) 用户文档

用户文档提供用户怎样使用程序的命令和指示，通常是指用户手册。好的用户文档就是联机帮助信息，用户可以在使用它时在终端上就获得必要的帮助和引导。

(3) 操作文档

操作文档指导用户如何运行程序，它包括操作员手册、运行记录和备用文件目录等。

(4) 数据文档

数据文档是程序数据部分的说明，它由数据模型和数据词典组成。数据模型表示数据内部结构和数据各部分之间的功能依赖性。通常，数据模型是用图形表示的。数据词典列出了程序使用的全部数据项，包括数据项的定义、使用及其使用地方。

(5) 历史文档

历史文档用于记录程序开发和维护的历史，但是不少人还未意识到它的重要性。历史文档包括三类，即系统开发日志、出错历史和系统维护日志。系统开发和维护历史对维护程序员是非常有用的信息，因为系统开发者和维护者一般是分开的。利用历史文档可以简化维护工作，如理解设计意图，指导维护程序员如何修改源代码而不破坏系统的完整性。



2.5 逆向工程和再工程

随着维护次数的增加，可能会造成软件结构的混乱，使软件的可维护性降低，束缚了新软件的开发。同时，那些待维护的软件又常是业务的关键，不可能废弃或重新开发。于是引入了软件再工程（Reengineering），即对旧的软件进行重新处理、调整，提高其可维护性，这种活动称为“软件再工程”，是提高软件可维护性的一类重要的软件工程活动。

再工程也称复壮（修理）或再生，它不仅能从已存在的程序中重新获得设计信息，而且还能使用这些信息来改建或重构现有的系统，以改善它的综合质量。一般软件人员利用再工程重新实现已存在的程序，同时加入新的功能或改善它的性能。

下面，讨论软件再工程的相关技术。

1. 逆向工程

软件的逆向工程是分析程序，力图在比源代码更高的抽象层次上建立程序表示的过程，是一个设计恢复的过程。使用逆向工程工具可以从已经存在的软件中提取数据结构、体系结构和程序设计结构。逆向工程的过程如图 2-7 所示。

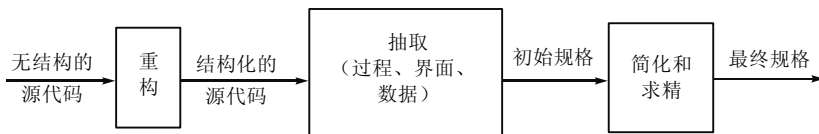


图 2-7 逆向工程的过程

逆向工程的过程从源代码重构开始,将无结构的源代码转换为结构化的源代码,提高了源代码的易读性。抽取是逆向工程的核心,内容包括处理抽取、界面抽取和数据抽取。处理抽取可在不同层次进行,如:语句段、模块、子系统、系统。使用逆向工程工具,可以从已存在程序中抽取数据结构、体系结构和程序设计信息。

出于法律约束的原因,公司一般只对自己的软件作逆向工程。通过逆向工程所抽取的信息,既可用于软件维护的任何活动,也可用于重构原系统,以改善它的综合质量。

2. 软件重构

软件重构是对源代码、数据进行修改,使其易于修改和维护,以适应将来的变更。通常,软件重构并不修改软件体系结构,而是关注模块的细节。

(1) 代码重构

代码重构的目标是生成功能相同,而质量更高的程序。由于需要重构的模块通常难以理解、测试和维护,因此,首先用重构工具分析代码,标注出需要重构的部分,然后进行重构,最后复审和测试重构后的代码,更新代码的内部文档。

(2) 数据重构

数据重构发生在较低的抽象层次上,是一种全局的再工程活动。数据重构通常以逆向工程活动开始,理解现存的数据结构,又称数据分析,再重新设计数据,包括数据标准化、数据命名合理、文件格式转换、数据库格式转换等。

(3) 软件重构的意义

提高软件质量和生产率,减少维护工作量,提高软件可维护性。

3. 再工程的成本/效益分析

再工程要耗费时间,占用资源。为了降低再工程的风险,必须进行成本/效益分析。Sneed提出了以下再工程的成本/效益模型:

① 与未执行再工程的持续维护相关的成本

$$C_{\text{maint}} = [p_3 - (p_1 + p_2)] \times L$$

② 与再工程相关的成本

$$C_{\text{reeng}} = [p_6 - (p_4 + p_5) \times (L - p_8) - (p_7 \times p_9)]$$

③ 再工程的整体收益

$$C_{\text{benefit}} = C_{\text{reeng}} - C_{\text{maint}}$$

式中, p_1 为当前某应用的年维护成本; p_2 为当前某应用的年运行成本; p_3 为当前某应用的年收益; p_4 为再工程后预期年维护成本; p_5 为再工程后预期年运行成本; p_6 为再工程后预期业务收益; p_7 为估计的再工程成本; p_8 为估计的再工程日程; p_9 为再工程的风险因子 ($=1.0$); L 为期望的系统生命期 (年)。

4. 再工程的风险分析

再工程与其他软件工程活动一样可能会遇到风险,软件管理人员必须在再工程活动之前



对其风险进行分析，对可能的风险提供对策。再工程的风险主要包括以下 3 个方面。

① 过程风险：在进行再工程活动中，对投入的人员缺乏管理，对再工程方案的实施缺乏管理，未作成本/效益分析。

② 应用领域风险：对再工程项目中的业务知识不熟悉，缺少专业领域的专家支持。

③ 技术风险：缺乏再工程技术支持，恢复设计得到的信息无用等。

此外，还有人员风险，工具风险等。

软件再工程是提高软件可维护性的一类重要的软件工程活动。与软件开发相比，软件再工程不是从编写规格说明开始，而是从原有的软件出发，通过再工程，获得可维护性好的新软件。

习题 2

2.1 为什么要进行软件维护？

2.2 怎样防止维护的副作用？

2.3 什么是软件可维护性？可维护性度量的特性是什么？

2.4 提高可维护性的方法有哪些？

2.5 软件的可维护性与哪些因素有关？在软件开发过程中应该采取哪些措施才能提高软件产品的可维护性？

第3章 软件工具与集成化环境



3.1 软件工具

工具在人类生活和生产活动中的地位和作用是众所周知的,它极大地提高了人类进行生产劳动的效率,促进了人类文明的发展。在软件的开发活动中,使用强大有力、方便适用的软件开发工具同样可以大大提高软件开发活动的效率和质量,促进软件产业的发展。

软件工具是用来辅助软件的开发、运行、维护、管理和支持等活动的软件系统,它的目的是降低软件开发和维护的成本,提高软件生产效率,改进软件产品的质量。

在软件工程活动中,软件工程师和管理员按照软件工程的方法和原则,借助于计算机及其软件工具的帮助,开发、维护、管理软件产品的过程,称为计算机辅助软件工程(Computer-Aided Software Engineering, CASE)。CASE的实质是为软件开发提供一组优化集成的且大量节省成本的软件开发工具,其目的是实现软件生命周期各环节的自动化并使之成为一个整体。

目前有两个层次的软件开发工具。一个层次是孤立的单个软件开发工具,用于支持软件开发过程中的某一项特定活动,这些零散的工具具有不同的用户界面、不同的数据存储格式,它们之间彼此独立,不能或很难进行通信和数据的共享与交换。显然,这种没有集成的软件开发工具难以有效地支持软件开发的全部过程,在软件开发的阶段,开发人员需要使用不同的工具,而这些工具之间又不能无缝的衔接,这种状况极大的限制了这些软件开发工具的效能的发挥。另一个层次的软件工具是集成化的CASE环境,它将在软件开发过程的不同阶段使用的工具进行集成,使其有着一致的用户界面和可以共享的信息数据库。

软件工具种类繁多、功能各异,根据它们在软件开发过程中的功能可以对其进行分类。这样的分类并不是绝对的,某些软件开发工具可能属于两种或两种以上的不同类型。按照软件活动的不同阶段,可以把软件工具分为:软件开发工具,软件维护工具和软件管理与支持工具。



3.1.1 软件开发工具

软件开发工具可以细分为:分析工具、设计工具、编码工具和调试工具。下面分别介绍这几类软件开发工具的具体作用。

1. 分析工具

分析工具是指用来辅助软件开发人员完成软件系统需求分析活动的软件工具。软件分析工具可以帮助系统分析人员根据需求的定义,生成完整、清晰、一致的功能规范。功能规范



是软件所要完成功能的准确而完整的描述,是软件设计者和实现者进行软件开发的依据。软件系统的功能规范应当准确、完整地陈述用户对软件系统的功能需求。软件分析工具主要包括三种类型:基于自然语言或图形描述的需求分析工具、基于形式化需求定义语言的工具和其他需求分析工具。

典型的有 Rational 公司的 Analyst Studio 成套的需求分析工具软件,这是用于应用问题分析和系统定义的一组相对很完备的工具集,适合于团队联合开发使用。Analyst Studio 包括以下内容。

- ◎ **Rational Requisite Pro:** 用来帮助开发人员在整个开发生命周期中创建与管理需求的一类需求管理软件。
- ◎ **Rational Rose Data Modeler Edition:** 使用工业标准的 UML,帮助开发者以图形方式交流与通信在软件总体结构中的各类需求。这是 Rational Rose 的专业版,在功能上组合了 Rational Rose Data Modeler Edition 软件的核心部分,还加入了 Data Modeler,支持数据库设计。
- ◎ **Rational Clear Quest:** 一个请求的变更管理系统,帮助开发团队根据发现缺陷和增强功能等请求进行跟踪并采取相应的行动。
- ◎ **Rational SoDA for Word:** 使文档资料的产生与管理自动化,并根据软件开发计划做出相应的有关报告。
- ◎ **Rational Unified Process:** 为软件工程定义作用和提供方向。

2. 设计工具

设计工具是用来帮助软件开发人员完成软件系统的设计活动的软件工具。软件开发人员使用设计工具,可以根据在软件的需求分析阶段获得的功能规范,生成与之对应的软件设计规范。软件系统的设计规范是对软件的组织 and 内部结构的描述。通常,设计规范分成概要设计规范和详细设计规范两个部分。其中,概要设计规范说明软件系统的功能模块结构和相互之间的调用与数据传输关系,而详细设计规范说明软件系统内部各个功能模块中包含的具体算法和数据结构。软件系统的设计规范对于软件开发活动具有十分重要的意义,它是软件开发人员进行程序编码实现的主要依据。目前,软件设计工具主要包括三种类型:基于图形描述、语言描述的设计工具,基于形式化描述的设计工具和面向对象的设计工具。

典型的有 Enterprise Architect,是一个基于 UML 的 Visual CASE 工具,主要用于设计、编写、构建和管理以目标为导向的软件系统。

3. 编码工具

编码工具主要包括:编辑程序、汇编程序、编译程序、调试程序等。

- ◎ 编辑程序主要完成程序代码的输入和编辑。任何一种文本编辑程序都可以用做程序的编辑程序。
- ◎ 汇编程序主要完成将汇编程序代码转化为功能等价的机器语言代码的工作。
- ◎ 编译程序主要完成将文本形式的源代码转化为功能等价的机器语言代码的工作。
- ◎ 调试程序主要是用于帮助程序员对程序中的错误进行发现和修改。

这些编码工具可能是一个集成的程序开发环境,其中集成了源代码的编辑程序、生成可

执行代码的编译程序和链接程序、用于源代码排错的调试程序及用于产生可供发布产品的发布程序。这些集成的程序开发环境的典型例子有：Microsoft 公司的 Visual C++、Visual Basic 和 Borland 公司的 Delphi、C++ Builder。另一种类型的编码工具并非一个集成的程序开发环境，其中的编辑、编译、链接等功能是由彼此独立的应用程序提供的，这些工具并没有被集成为一个统一的开发环境和用户界面。这方面的典型例子有：Sun 公司的 JDK 开发工具。

4. 调试工具

调试工具也称为排错工具，主要用于在程序编码过程中，及时地发现和排除程序代码中的错误和缺陷。调试工具主要分为：源代码调试程序和调试程序生成程序两类。

(1) 源代码调试程序

源代码调试程序用于帮助程序开发人员了解程序的执行状态和查询相关数据信息，发现和排除程序代码中存在的错误和缺陷。这一类工具一般由执行控制程序、执行状态查询程序和跟踪包组成。执行控制程序可用于断点定义、断点撤销、单步执行、断点执行、条件执行等功能。执行状态查询程序用于帮助程序员了解程序执行过程中 CPU 寄存器、堆栈、变量和其他数据结构中存储的数据与信息。跟踪包用于跟踪程序执行过程中所经历的事件序列。通过对程序执行过程中各种状态的判别，程序员可以进行程序错误的识别、定位和纠正，完成程序的调试工作，确保软件产品的质量和可靠性。

(2) 调试程序生成程序

调试程序生成程序是一种通用的调试工具，能够针对给定的程序设计语言，生成一个相应的源代码调试程序。在早期的程序开发过程中，编码工具和调试工具没有被集成在一起，而是作为两个独立的软件提供给软件开发人员使用。编码工具一般采用通用的文本编辑软件，而调试工具是由操作系统提供的，与具体的程序设计语言无关。软件开发人员通常先用编码工具完成源代码程序的编辑、修改，再调用编译器完成对源代码的编译，接着调用调试工具发现程序中的错误和缺陷，最后再次使用编码工具完成对程序中错误的修改和纠正。

5. 软件开发工具的评价与选择

在软件开发过程中，选择理想的开发工具对于提高软件开发的质量和效率，降低软件开发的成本具有十分重要的意义。通常，可以根据以下几个标准来评价一个软件开发工具的优劣程度。

(1) 功能

软件开发工具具有完备的开发功能是最基本的要求。软件开发工具不仅要实现所要求的功能，支持用户所采用的开发方法，还应该一些有用的辅助功能，例如自动保存、语法检查等。

(2) 硬件要求

软件开发工具自身也是一个运行在硬件平台上的软件程序，它的运行需要占用一定的存储资源和计算资源。一个硬件要求较低的开发工具可以为开发人员节省相应的硬件开销和开发成本。

(3) 性能

软件开发工具的运行速度等性能指标将直接影响工具的使用效果。使用一个具有较高的



运行速度的软件开发工具可以有效地提高软件开发的速度和效率。

(4) 方便性

软件开发工具应该具有十分友好的用户界面，方便用户的使用。软件开发工具的界面应能裁剪和定制，以适应特定用户的需要，提供简单有效的执行方式；同时，软件开发工具应能及时检查和发现用户的操作错误。

(5) 服务和支持

为了适应软件开发技术的迅猛发展，软件开发工具需要不断地进行升级和改进。同时，不同于普通的软件产品，软件开发工具的功能强大、使用复杂，对使用者有较高的要求。因此，软件开发工具的生产厂商应该为工具提供有效及时的技术服务和支 持，例如，软件使用的培训和咨询、软件版本的更新、错误和缺陷的及时修复，同时还应该提供关于软件开发工具的齐全详尽的文档。

3.1.2 软件维护工具

众所周知，软件维护是整个软件工程中一个重要的环节和任务，其主要任务是在软件产品投入运行以后，纠正软件开发过程中未发现的错误，改进和完善软件的功能和性能，以适应用户的需求，延长软件产品的使用寿命，使其创造更多的价值。软件维护工具可以帮助软件维护人员完成对软件产品的维护工作。重要的软件维护工具包括：版本控制工具、文档管理工具、开发信息库工具、逆向工程工具、再工程工具。

(1) 版本控制工具

在软件开发过程中，对软件系统进行变动和修改是不可避免的。因此，对于一个软件产品会形成新旧不同的若干个版本。显然，对软件产品的不同版本进行控制和管理是十分重要的。版本控制工具用于帮助软件维护人员实现对软件版本的存储、更新、恢复和管理。版本控制工具一个典型代表是 UNIX 操作系统的 SCCS（源代码控制系统）。SCCS 为一个源代码文件的所有版本建立一棵版本树，该文件的每一个版本都是该版本树上的一个结点。SCCS 完整存储该文件的第一个版本，而对于后续的其他版本则只存储它与以前版本的不同之处。利用这种管理方式，SCCS 可以通过版本树维护各个版本的更新历史，并允许恢复到以前的任何一个版本。

(2) 文档管理工具

文档也是软件开发过程中的重要产品。在许多软件的开发过程中，都要花费大量的人力和物力来开发和组织文档，通常，软件开发组织要花 20%~30% 的工作量来完成软件文档的编写。因此对软件开发过程中产生的文档进行管理和维护对提高软件开发的质量和效率具有重大的意义。文档管理工具用于对软件开发过程中产生的文档进行分析、组织、维护和管理。例如，基于数据流图的需求文档管理工具在对数据流图中的某些成分进行分析时，可以确定该成分的影响范围和被影响范围，以帮助开发成员在对该成分进行修改时，确定其影响范围内的其他成分是否也需要进行相应的变更。而针对源程序文档的文档管理工具可以帮助编码人员确定其中的全局变量或数据结构的作用范围。

(3) 开发信息库工具

开发信息是指在软件系统的开发过程中,用来维护软件项目开发的相关信息,例如,程序中的对象、模块等内容。开发信息库工具可以用于记录每个对象的开发与修改信息;维护对象与相关信息之间的关系,包括记录对象的开发人员、新版本对象中发生的改动、对象中存在的错误、对该对象进行测试时使用的测试用例、测试结果之间的关系等内容;还可记录用来生成此软件产品的所有开发工具的版本信息、所采用的程序设计语言和应用程序开发接口。

(4) 逆向工程工具

软件的逆向工程是指对已有的程序进行分析,以获得比源代码更高级的表现形式,是一个设计恢复的过程。逆向工程工具可以帮助软件维护人员从已存在的程序中提取出数据结构、体系结构、程序总体设计等各种有用的软件开发信息。诸如反汇编工具、反编译工具等早期的逆向工程工具,用于将机器代码转换成汇编语言或高级程序语言的代码,以方便开发人员对代码的阅读、理解和修改。现在的逆向工程工具能够分析高级程序设计语言的源程序,恢复程序的控制结构、流程图、PAD图等更高级的抽象信息,为软件的理解和维护提供方便。

(5) 再工程工具

软件系统的再工程是指在获得软件设计信息的同时,利用这些信息修改或重构软件系统的工作。根据用户的需求,软件开发人员可利用再工程重新实现已有的软件系统,同时增加新的功能和改进性能。

再工程工具可以用来辅助软件开发人员重构一个功能和性能更为完善的软件系统。目前,再工程工具的使用主要集中在代码重构、程序结构重构和数据结构重构等方面。数据结构重构是指通过对数据描述的分析,重新构造出新的数据结构;程序结构重构是指将一个非结构化或结构化程度比较低的源程序改造为一个等价的高度结构化的程序;代码重构是指把一种程序语言书写的程序转换成功能等价的但由另一种程序语言书写的或适用于不同硬件平台的程序,例如,将由C语言书写的程序转换为用Java语言书写的功能等价的应用程序。

3.1.3 软件管理与支持工具

在软件系统开发的各个阶段都涉及软件产品的管理与支持。软件管理与支持工具用于辅助软件管理人员和软件支持人员对软件系统的管理和支持活动,以确保软件产品的质量和软件产品的开发效率。在软件管理与支持工具中,比较重要的包括:软件评价工具、配置管理工具、项目管理工具、风险分析工具。

(1) 软件评价工具

软件评价工具的主要作用是帮助软件产品的管理与支持人员对软件产品的质量进行评价。软件评价工具对于实现对软件产品的质量的控制,确保软件产品的正确性、可靠性,具有十分重要的意义。根据某个软件质量模型,例如,ISO软件质量度量模型、McCall软件度量模型,软件评价工具可以对软件产品的质量加以度量,然后根据相应的评价结果形成对该软件产品的质量评价报告。对一些已经量化的度量指标,可以利用软件评价工具自动获取。但是,在目前的软件质量度量活动中,许多度量的指标都还不能加以定量的描述,通常采用的方式是,通过相关的专家以人工的方式为软件产品的某些度量指标给一个评分,然后再由



软件评价工具根据输入的评分值,形成对该软件产品的质量评估报告。除此之外,有些软件评价工具还可以分析被评价的软件系统的程序结构,然后根据某种软件复杂性模型,实现对软件系统的复杂性度量。

(2) 配置管理工具

在软件产品的开发过程中,对其进行变动和修改是不可避免的。而这些变动和修改很可能在软件开发人员之间导致混乱和误会,严重影响软件产品的开发质量和开发效率。因此在对软件产品进行修改之前,必须进行相应的分析论证,以确保修改的质量和正确性,在修改之后必须进行记录,并将修改通知相关人员。为了减少由于修改而导致的混乱,提高软件产品的开发效率,提出了软件配置管理这一概念。软件配置管理是对软件修改进行标示、组织和控制的技术,用来协调和控制软件开发的整个过程。软件配置管理是保证软件质量的重要一环,用于控制软件的修改和变动,其具体任务包括:标示软件配置中的各个项目、控制软件的各种版本、控制对软件的修改、审计配置、报告配置状况等内容。

综上所述,软件配置管理工具的主要作用是帮助软件配置管理人员完成软件配置项的标示、版本控制、审计和状态统计等基本任务,使得各配置项的访问、修改易于实现,达到简化审计过程、改进状态统计、减少软件错误、提高软件质量的目的。

(3) 项目管理工具

软件项目管理的主要任务是制定软件开发计划,跟踪、监督和协调软件开发的进度,以保证软件产品能够按时保质完成。软件项目管理是软件工程中保护性和支持性的活动,它开始于软件开发之初,贯穿整个软件产品定义、开发和维护的全过程。项目管理的具体内容是对软件项目所涉及的人员、费用、进度和质量4个方面的有效管理。

软件项目管理工具用来帮助项目管理人员对软件产品的开发活动进行有效的管理。例如,成本估算工具,就是根据某一种成本估算模型,如 Halstead 模型、Putnam 模型、COCOMO 模型等,对软件项目的成本进行估算。成本估算工具通过对软件产品间接的测量,例如,对代码行、功能点的测量,来估算软件项目的规模,并描述总的项目特征,如问题的复杂度等,然后按一定的估算模型计算出整个软件项目的工作量、开发周期和开发人员的数量等结果。除此之外,当软件项目在开发过程中发生改变时,成本估算工具还可以计算出其对整个开发成本的影响。

(4) 风险分析工具

标示潜在的风险并设计相应的计划去缓解、监控和管理风险,这对于一个大型项目是极为重要的。风险分析工具可以通过提供对风险的标示和分析的详细指南,使得项目管理者能够有效地对软件项目开发过程出现的风险进行控制和规避。



3.2 集成化CASE环境



3.2.1 概述

上一节中所介绍的各种软件开发工具在软件产品的某个开发阶段具有重要的作用,但是它们彼此之间是相互独立的,有着不同的用户界面、不同的数据存储格式,不能够有效地进

行相互通信和数据共享,这些缺陷极大地限制了这些软件开发工具最大效能的发挥。因此,集成这些孤立的软件开发工具,为软件开发人员提供有着统一标准、统一数据存储方式和集中数据库的集成化的软件开发环境,对于提高软件开发的质量和效率是十分必要的。集成化 CASE 环境的基本含义是,将多个 CASE 工具结合起来,使得各种软件开发信息能够在不同 CASE 工具之间、不同开发阶段之间以及不同开发人员之间顺畅的传递。

通常,一个优秀的集成化 CASE 环境一般应该满足下列基本要求和条件:

- ① 能够有效地帮助各类软件开发人员进行快捷、顺畅的信息交流;
- ② 能够让软件开发人员以任意合理的顺序使用集成环境中的各种 CASE 工具;
- ③ 集成环境中的各种 CASE 工具应该提供一个风格一致的用户界面以方便软件开发人员的使用;
- ④ 能够为集成环境中的各种 CASE 工具提供共享各种软件开发信息的功能和机制;
- ⑤ 在某项软件开发信息发生修改或调整时,集成环境能够跟踪并确认该修改所导致的影响传播范围;
- ⑥ 集成环境能够为所有的软件开发信息提供版本控制等配置管理功能。

相对于独立的软件开发工具,集成化的 CASE 环境可以为软件的开发带来如下的好处:

- ① 软件开发过程中的所有信息都采用统一的存储格式,集中统一存储在共享的中心数据库中,使得软件工具之间、开发人员之间、开发活动的各个过程之间可以方便而高效的进行数据的共享和交换;
- ② 集成化的 CASE 环境采用统一的用户界面,为软件开发人员提供了更为方便的使用平台,并且改善了开发人员之间的协调能力;
- ③ 集成化的 CASE 环境的使用可以贯穿软件开发的各个阶段,包括分析、设计、编码、测试、维护和配置,这使得软件开发活动和相关的开发信息可以很流畅地由一个开发阶段过渡到下一个开发阶段。
- ④ 集成化的 CASE 环境也具有更好的可移植性,使其可以适用于不同的硬件平台和操作系统。

在软件工具的发展过程中,出现了许多集成化的 CASE 环境,这些开发环境的集成度是不一样的。有些开发环境的集成仅仅在不同的工具之间提供了一个信息交换的途径,有些开发环境的集成仅限于为不同的开发工具提供一致的用户界面,有些开发环境则提供了统一的用户界面、统一的信息存储。按照集成度的高低,可以将集成化的 CASE 环境大致划分为以下 3 种层次。

(1) 具有信息传递的软件工具集

具有此种集成度的开发环境中,工具之间是完全独立的,集成度非常低。它们之间有着不同的用户界面和信息存储格式。这些工具的信息使用不同的格式进行存储,它们借助于操作系统的文件服务和数据交换服务,使得工具 A 的输出文件能够被导入到工具 B 中,实现不同工具之间的数据交换和共享。

具有信息传递的软件工具集如图 3-1 所示。

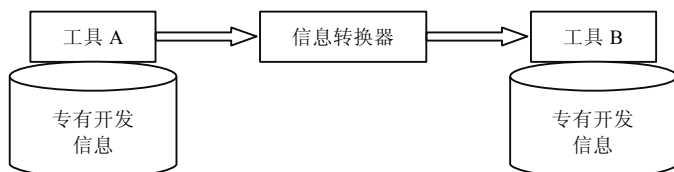


图 3-1 具有信息传递的软件工具集

不难看出,这种集成方式的集成度是非常低的。在不同的文件格式之间进行数据的导入/导出的效率是很难令人满意的,这种信息交换会花费大量的时间。同时随着开发工具的数量增加,每种开发工具使用不同的文件格式进行信息的存储,文件格式之间转换将变得非常复杂。并且,反复地进行格式转换可能导致文件信息的一致性和完整性遭到破坏。

(2) 具有公共界面的软件工具集

这些软件工具集为用户提供了一致的公共用户界面和操作方式,例如,这些工具有着相同的菜单、工具按钮、快捷方式,这为软件开发人员提供了极大的便利。但是,具有此种集成度的软件工具之间的数据交换仍然沿用不同格式的文件导入/导出的方式,这严重地影响了彼此之间数据交换的效率和数据的完全性与完整性。具有公共界面的软件工具集如图 3-2 所示。

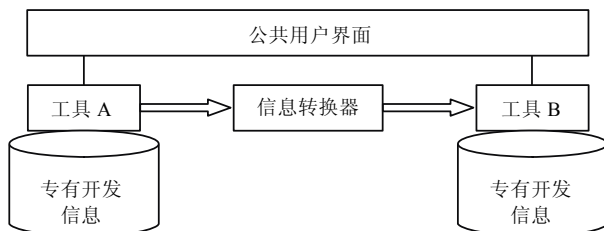


图 3-2 具有公共界面的软件工具集

(3) 信息共享的软件工具集

具有此种集成度的开发环境中,工具之间不仅具有一致的用户界面和操作方式,而且对不同工具的开发信息进行统一的存储和管理。这种信息共享的集成方式从根本上解决了在不同的软件工具之间进行信息交换的问题,提高了工具之间的继承度。同时它也要求不同的软件工具之间要遵守共同的信息存储的标准。信息共享的软件工具集如图 3-3 所示。

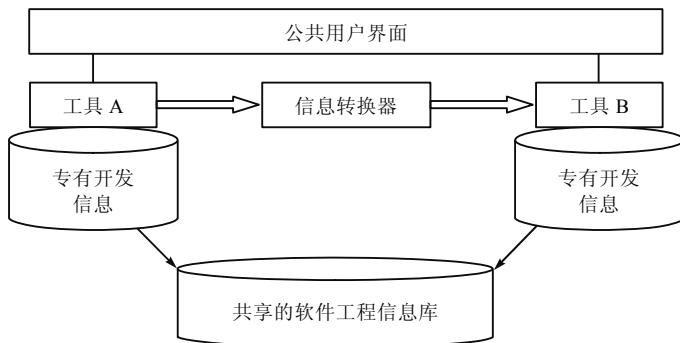


图 3-3 信息共享的软件工具集

3.2.2 集成化的CASE开发环境的要求

一个集成化的 CASE 开发环境通常需要完成以下三个层次的集成。

1. 平台的集成

平台是指 CASE 工具运行所处的计算机硬件和操作系统环境。在计算机长期的发展过程中,出现了许多不同类型的计算机硬件体系结构和操作系统,它们在现在和将来会长期并存。在一个组织机构内部经常会同时使用这些有着显著差异的计算机硬件和操作系统,这时,为了实现运行在不同平台之上的 CASE 工具之间的互操作和数据共享,基于平台的集成就成为了一个急需解决的问题。

2. 界面的集成

界面集成是指集成化的 CASE 环境中的各种软件开发工具采用统一的用户交互界面和方式,以达到降低用户学习和使用这些工具的难度和复杂度的目的。

界面集成可以分为以下三个层次。

(1) 窗口集成:被集成的不同的软件工具采用相同的窗口外观,包括相同风格的菜单、对话框、工具栏等外观特征。

(2) 命令集成:不同的软件工具对相似的功能采用相似的命令。许多软件开发工具中的许多操作往往可以通过命令行或者命令的快捷方式来完成。命令的集成是指完成相似命令的语法和参数在不同软件开发工具之间应该是相同或相似的。

(3) 交互集成:不同的软件开发工具对相同的功能采用统一的交互方式。

3. 数据的集成

为了实现不同的 CASE 工具之间的数据交换与共享,数据的集成是集成化的 CASE 开发环境需要解决的一个核心问题。

目前有着三种不同类型的数据集成,它们采用了不同的数据存储方式。

(1) 共享文件

被集成的各种 CASE 工具约定采用统一的文件格式来存储各种软件开发信息。借助于操作系统所提供的文件共享功能实现不同 CASE 工具之间对数据的共享和交换。

这种数据集成方式所采用的文件格式常用的是纯文本文件,由于其中不包含与格式有关的信息,所以它可以提供一种非常原始和简单的数据共享的媒介。另外一种文件格式就是 XML(可扩展置标语言)文件。由于 XML 是一种与软硬件平台无关、与应用无关的结构中立的置标语言,因此它为不同平台之间、不同应用程序之间信息的交换与共享提供了一种良好的数据格式。并且,XML 还具有很好的可扩充性,它的标志不是固定不变的,可以根据实际需要定义新的标志。在集成化的 CASE 环境中,只要不同的软件开发工具之间约定采用统一类型的 XML 文档,那么它们之间的数据集成就可以得到解决。

(2) 共享数据结构

这里所说的数据结构是指包含软件开发信息的实体关系图、数据流图等相关数据实体。集成的 CASE 环境中的不同软件开发工具之间共享这些相关的数据结构以实现对数据的集



成。这种方式的数据集成可以提供对开发信息一致、灵活的存储和表示，但是它的开放性和适应性有一定的缺陷。主要原因是，这些共享的数据结构比较复杂，并且它的内部格式一般都是软件工具生产厂商私有的，第三方的开发商很难知道其中的细节，因此他所开发的新的软件工具不能加入到已有的集成 CASE 环境中。所以这种基于共享数据结构的 CASE 环境具有一定的封闭性，很难在其基础上进行二次开发或者加入新的软件开发工具。

(3) 共享数据库

围绕共享的数据库管理系统的集成是灵活性最好、适应性最强的数据集成方式。为了实现数据的集成，需要对软件开发过程中产生的各类信息进行抽象，采用面向对象的方法可以把所有的软件开发信息都抽象为一个对象。由一个公共的对象管理系统（OMS）来统一管理这些信息对象和软件开发工具之间的操作和联系。各种软件开发工具通过统一的接口访问数据库中的软件开发信息，以实现不同软件开发工具之间的数据集成。

3.2.3 集成化的CASE开发环境的体系结构

通常，一个集成化的 CASE 开发环境的体系结构一般需要由以下 4 个部分构成，它们分别是：一个共享的软件工程信息库，一个管理开发信息的对象管理层，一个协调各个 CASE 工具的工具集成层，一个提供统一用户接口的用户界面层，如图 3-4 所示。

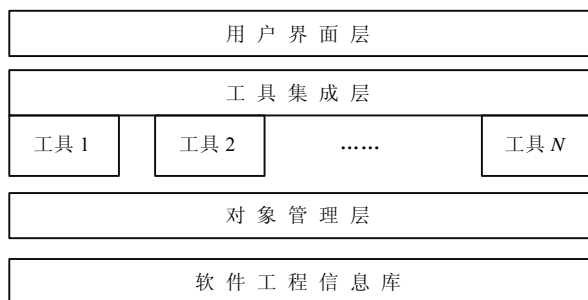


图 3-4 集成化的 CASE 开发环境的体系结构

1. 用户界面层

用户界面层主要由一个标准化的统一的界面工具箱和各个 CASE 工具所共同遵守的界面协议所组成。其中，界面工具箱由窗口、菜单、对话框、按钮等用户界面元素以及诸如事件触发、消息传递等界面元素控制机制共同组成。而诸如用户界面布局的约定、菜单的命名与组织规则等约定与规范构成了不同工具所共同遵守的界面协议。

在设计用户界面层时，通常要实现两个目标，即

- ◎ 不同工具的用户界面需要进行集成，统一提供给用户，而不是分离的多个用户界面；
- ◎ 不同工具的用户界面需要有统一、一致的外观和操作方式。

2. 工具集成层

工具集成层主要完成对构成集成 CASE 开发环境的所有工具的管理和协调任务。在开发人员使用集成 CASE 开发环境进行软件开发时，经常需要同时使用多个 CASE 工具。此时，在不同的 CASE 工具之间进行任务的组织和协调，实现不同 CASE 工具之间开发信息

的传递和共享,实施安全检查和审计功能是十分重要的,而这也正是工具集成层完成的任务和功能。在此基础上,工具集成层还能够收集各种 CASE 工具在软件开发过程中的使用情况,并对相应的使用数据进行汇总、分析,以便改进软件开发方法,提高集成 CASE 开发环境的运行效率。

为了实现对不同 CASE 工具的管理与协调任务,工具集成层通常具有执行机制和通信机制。

执行机制是指集成 CASE 开发环境能够根据具体的软件开发事件触发相应的 CASE 工具的启动与运行。

通信机制是指集成 CASE 开发环境能够在不同的 CASE 工具之间传递软件开发信息。

需要指出的是,无论是执行机制还是通信机制,通常都需要集成 CASE 开发环境运行所在的操作系统平台的支持。因此,为了提高集成 CASE 开发环境相对于操作系统平台的独立性,应该考虑在集成 CASE 开发环境与操作系统之间设置隔离层,以实现集成 CASE 开发环境的可移植性。

3. 对象管理层

对象管理层主要完成在集成 CASE 开发环境中对软件开发信息的管理和集成工作,实现 CASE 工具与信息的集成以及信息与信息的集成。在软件开发的过程中,信息的种类繁多复杂。为了对这些种类繁多的信息进行管理和集成,对信息加以抽象是必需的。面向对象的方法是一种比较常见的信息抽象方式。通过把信息抽象为对象以后,信息与 CASE 工具之间的集成可以通过对这些信息对象的相应操作加以完成。

另外,信息与信息的集成是指集成 CASE 开发环境能够表示、维护相关的开发信息并利用信息之间的关系完成相应的操作,例如,对信息项的完整性、一致性的检查,确定信息项修改所影响的范围等。

4. 软件工程信息库

在整个集成 CASE 开发环境中,软件工程信息库处于一个核心地位,是其他层次模块的基础。

(1) 软件工程信息库提供的服务

一个合理的软件工程信息库需要向用户提供两种不同类型的服务:

① 普通数据库管理系统的服务

软件工程信息库首先需要作为一个普通的数据库管理系统,完成对软件开发过程中产生和出现的各类信息与数据的存储、查询、修改等操作。

② 特定于集成 CASE 开发环境的服务

除了具备普通的数据库管理系统的大部分基础功能以外,为了给集成化开发环境提供良好的信息服务,软件工程信息库还需要提供的特殊服务,包括:

- ◎ 信息的完整性;
- ◎ 信息与工具的集成;
- ◎ 信息与信息的集成;
- ◎ 文档规范化;



◎ 版本控制与配置管理。

(2) 软件工程信息库的内容

软件工程信息库作为整个集成 CASE 开发环境的核心，其中存储着软件系统的相关开发信息，具体地说，包含以下 5 类信息。

① 企业信息

企业信息主要包括：业务域分析、业务功能分析、业务规则分析、过程模型（场景）、信息体系结构等内容。

② 项目管理信息

项目管理信息主要包括：项目计划、项目分解结构、项目成本估算、项目进度安排、项目审计信息等内容。

③ 系统文档

系统文档主要包括：项目需求文档、软件用户手册等内容。

④ 测试和评估信息

测试和评估信息主要包括：项目测试计划、项目测试数据用例、项目测试结果、软件质量度量、项目统计分析等内容。

⑤ 设计开发信息

设计开发信息主要包括：项目数据结构定义、项目过程定义、项目类定义、项目算法、项目命名标准等内容。

(3) 软件工程信息库的特征和功能

支持软件开发信息维护与管理的软件工程信息库应该具备的特征和功能如下。

① 非冗余数据的存储

冗余数据的存储不仅会导致存储空间的浪费，而且会给数据的一致性检查、版本控制和配置管理带来很大的困难。因此，在软件工程信息库中要求每一个数据对象仅存储一次，可以被所有需要该对象的 CASE 工具所访问。

② 高层的抽象访问

软件工程信息库需要为 CASE 工具提供对数据对象的公共访问机制，使得不需要在每个 CASE 工具中重复设置数据处理操作。

③ 数据的独立性

软件工程信息库应当把数据对象的存储与物理介质隔离开来，使得当配置发生改变时，数据的存储不会受到相应的影响。

④ 事务控制机制

为了在多个开发人员和多个 CASE 工具同时访问软件工程信息库时维护数据的一致性和完整性，软件工程信息库需要提供事务控制机制。通常，事务机制通过对数据记录加锁、分阶段提交对数据库的修改、事务日志及基于日志的数据恢复等手段来实现在并发访问时，对数据一致性、完整性的控制。

⑤ 数据的安全性

为了实现对数据的安全访问与存储，软件工程信息库通过使用口令与授权、数据备份和恢复等手段可以有效地确保数据的安全性。

⑥ 开放性

为了确保软件工程信息库与外界的数据交流和传递,软件工程信息库通常需要提供数据的导入、导出功能。

⑦ 多用户支持

一个功能完善的软件工程信息库应该允许多个开发人员同时使用,因此,软件工程信息库必须通过访问仲裁和记录加锁等手段来管理多个 CASE 工具或多个开发人员对数据对象的并发访问。

⑧ 友好的用户界面

为了方便用户的使用,提高软件开发的效率与质量,软件工程信息库需要为软件开发人员访问使用数据对象提供友好、一致的用户界面。

(4) 软件工程信息库的实现方法

为了将软件开发过程中产生的各种纷繁复杂的开发信息与数据在软件工程信息库中进行统一的存储和管理,通常的做法是采用面向对象的基本思想,将所有需要存储的软件信息项均视为对象,同时配置相应的元模型管理机制和 CASE 工具的触发控制机制。

为了对各种软件开发信息项进行元级的描述,需要为软件工程信息库配置元模型管理机制,其主要作用包括:

- ◎ 描述各信息项的定义和属性,例如,信息项的类型、属性、表示方法,产生该信息项的 CASE 工具,使用该信息项的 CASE 工具等;
- ◎ 描述信息项之间的相互关系;
- ◎ 描述软件开发过程中的工作流程和事件;
- ◎ 描述软件设计规则。

而 CASE 工具的触发控制机制是指,当软件工程信息库中某些软件开发事件发生时,能够将此消息通知相关的 CASE 工具,并触发相关的 CASE 工具对发生的事件进行响应和处理。例如,当软件工程信息库中的某个软件信息项发生修改时,这时 CASE 工具的触发控制机制就会将该修改通知配置管理工具,并触发配置管理工具对该修改的数据一致性进行检查。

软件工程信息库中除了具备普通数据库管理系统的基本功能以外,还有许多特殊的功能,是普通数据库管理系统所无法支持和提供的。这些特殊的功能说明如下。

① 复杂数据结构的存储

在软件工程信息库中,除了要存储简单的数据元素外,还要存储图、文档等复杂的数据类型。为了实现对这些复杂数据结构的存储,软件工程信息库运用面向对象的基本思想,将所有数据视为对象,借鉴面向对象数据库的组织方式,采用元模型来描述存储在其中的复杂数据的结构、关系与语义信息模型。同时,元模型还必须能够扩展,以适应新出现的数据形式。

② 数据完整性的实施

为了在多个用户、多个 CASE 工具同时访问软件工程信息库中的软件开发数据时,有效地确保数据的完整性和一致性,软件工程信息库中的元模型通常包含一组信息完整性规则。每当软件工程信息库中一个数据项发生修改时,一种被称为“触发器”的机制将被启动,



以完成对相关数据的完整性检查。

③ 语义丰富的操作接口

软件工程信息库中存储的软件开发信息需要被多个 CASE 工具存储和访问,这要求这些开发信息必须采用独立于 CASE 工具的方式进行存储,同时这些信息应该通过一个统一的并且语义丰富的操作接口被集成开发环境中的其他 CASE 工具所访问和使用。通过这种方式,来实现不同 CASE 工具对数据信息的共享,实现集成 CASE 开发环境对数据的集成。

④ 过程/项目管理

软件工程信息库中不仅存储有关软件应用本身的信息和目标软件的开发信息,而且还要包含关于每个特定项目的特征,例如,项目规划、可用资源清单等,这为通过使用软件工程信息库中的数据自动地协调软件开发活动和项目管理活动提供了可能。因此,可以设计一些专门的 CASE 工具,帮助软件开发和管理人员完成调整软件开发进度、生成各类分析报表、自动任务分派等工作。

⑤ 版本控制

随着软件开发活动的进行,特别是在由大量开发人员共同进行的一个开发活动中,各种软件产品和开发信息可能会出现许多不同的版本。软件工程信息库应该存储所有开发信息的全部版本,使得软件开发人员可以有效地管理这些开发信息,并允许开发人员在测试和调试过程中针对某个特定的开发信息回到其以前的任何版本。在软件工程信息库中存储着包含图形、文本、外部文件在内的许多复杂数据结构,而这些数据结构本身又是由若干更小的数据项复合而成的。因此,在进行开发信息的版本控制时,要考虑信息对象的粒度。一个成熟的软件工程信息库应该能够以任何粒度记录、跟踪、控制信息对象的版本。例如,对一个复杂数据结构和单个简单数据元素的版本控制。

⑥ 变更的跟踪和管理

软件工程信息库中存储的信息、数据之间不是彼此独立的,而是存在着大量的相互关系,通常,这些开发信息之间的关系在软件系统的整个开发过程中是基本不变的。一个完善的软件工程信息库需要提供对数据之间关系进行管理的机制和功能,这被称为链接管理。显然,链接管理所提供的数据之间关系的管理功能对于维护软件工程信息库中数据信息的完整性和一致性是至关重要的。其中最重要的是,当数据信息发生变更时,确定该变更的影响传播范围。例如,当某个数据流图发生修改时,链接管理可以检测确定相关的数据字典、代码实现是否也需要进行相应的修改。

⑦ 需求跟踪和管理

需求跟踪和管理是依赖于链接管理的一个特殊功能,其主要任务和作用是:指明设计文档和最终软件产品的各个部分是用来实现和完成哪些用户需求的。显然,借助于链接管理功能,利用需求规格说明、设计文档、源代码程序可以实现对用户需求的跟踪和管理,并在用户需求发生变更时,可以准确地确定这个需求变更对软件设计和实现的影响范围与程度。

⑧ 审计跟踪

审计跟踪的主要功能和任务是:在软件工程信息库中信息项发生变更时,记录变更发生的时间和变更的施加者。这些关于变更的相关信息可以作为特定对象的属性存储在软件工程

信息库中。显然, 审计跟踪对变更的记录对于维护软件开发信息的完整性、一致性具有十分重要的意义。



3.3 软件开发工具 Rational Rose



3.3.1 Rose 工具简介

Rational Rose 是 Rational 公司出品的基于 UML 的功能强大的可视化建模工具, 它可以与多种开发环境无缝集成并支持多种开发语言, 其中包括: Visual Basic、Java、PowerBuilder、C++、Ada、Smalltalk、XML DTD 等。可以运行 Rational Rose 的系统平台包括了目前大多数的主流操作系统, 如: Windows 9X、Windows 2000、Solaris、AIX 和 HP-UX 等。利用 Rose 可以开发出几种不同的模型图, 用以在不同的开发阶段、从不同的方面为软件系统的开发建立模型。

这些模型图包括: 业务用例图 (Business Use Case Diagram)、用例图 (Use Case Diagram)、类图 (Class Diagram)、协作图 (Collaboration Diagram)、时序图 (Sequence Diagram)、活动图 (Activity Diagram)、状态图 (Statechart Diagram)、构件图 (Component Diagram) 和部署图 (Deployment Diagram)。

Rational Rose 不仅拥有强大的功能, 而且具有方便友好的用户界面, 可以帮助软件开发人员进行高效的软件开发。Rational Rose 的用户界面如图 3-5 所示。

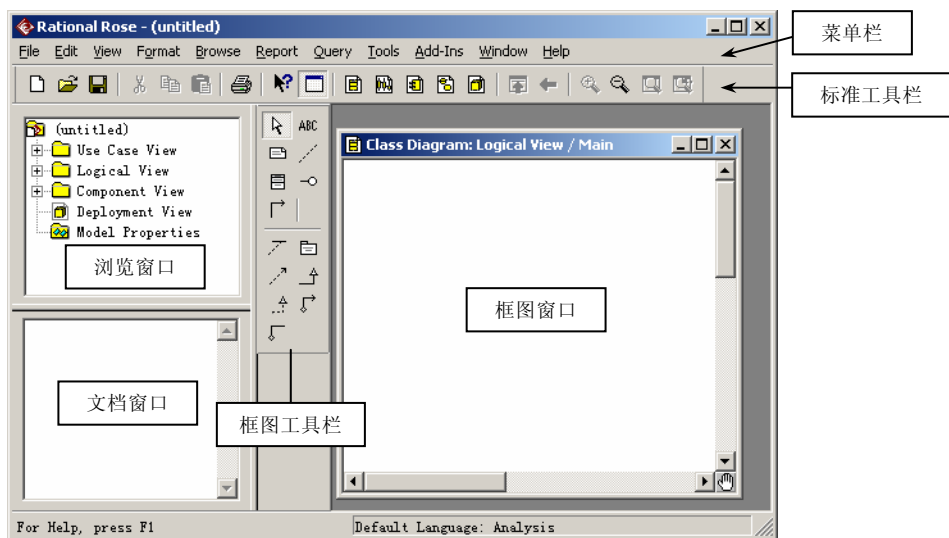


图 3-5 Rose 的用户界面

Rose 的用户界面包括以下几个部分。

- (1) 菜单栏: 包含了所有的 Rose 命令和操作。
- (2) 标准工具栏: 用于快速访问 Rose 中的常用命令和操作。
- (3) 浏览窗口: 采用树状层次结构, 用于在 Rose 模型中进行浏览。通过浏览窗口可以



快速地访问到 Rose 模型中的各个模型元素。

(4) 文档窗口：用于为模型元素建立说明文档。

(5) 框图工具栏：用于在模型图中添加各种模型元素，其内容随打开的 UML 模型图的类型不同而有所不同。

(6) 框图窗口：用于显示和编辑 Rose 模型中的各种 UML 模型图。当增删框图窗口中的模型元素时，Rose 会自动更新浏览窗口中的内容；同样当修改浏览窗口中的模型元素时，相应的修改也会自动反映在框图窗口中。



3.3.2 业务用例图

业务用例图（Business Use Case Diagram）用于建立机构的业务模型，包括描述整个机构业务执行的流程和所提供的功能等内容。其中涉及的模型元素有：业务执行者、业务工人、业务用例、业务实体和机构单元。业务用例图描述了这些元素及其相互关系。

如图 3-6 所示，在浏览窗口中，右键单击“Use Case View”项目，弹出快捷菜单，选择菜单项“New►Use Case Diagram”，就可以为模型创建新业务用例图。

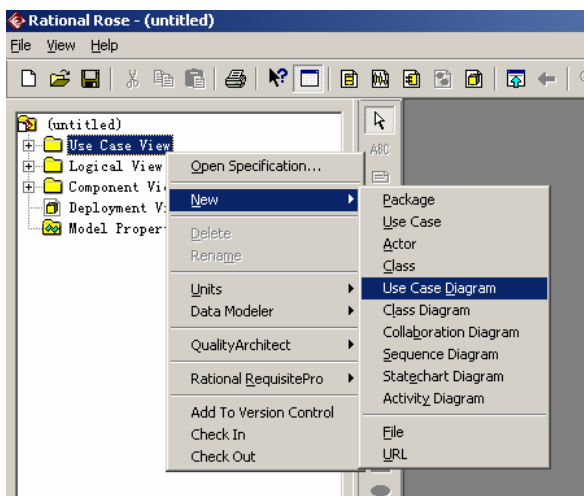


图 3-6 创建新业务用例图

业务用例图包括以下几类模型元素。

① 业务用例：表示机构中一组业务的执行和 workflows。

② 业务执行者：表示处于机构之外，与机构进行交互的实体。

③ 业务工人：表示处于机构之内，参与了业务执行流程的角色。

④ 业务实体：表示机构业务流程中需要使用的物理实体，例如，资金账目、客户订单、客户资料等。

⑤ 机构单元：表示业务工人、业务实体和其他相关模型元素的集合，是组织业务模型的机制。

在业务用例图中，模型元素之间存在以下两种关系。

(1) 关联关系描述业务执行者或业务工人与业务用例之间的通信和联系。Rose 中使用带箭头的线段表示各模型元素之间的关联关系,箭头的方向从通信的发起者指向通信的接收者。如图 3-7 所示,描述了一个图书销售管理系统中各个模型元素之间的关联关系。

(2) 泛化关系描述模型元素之间抽象与具体、一般与特殊的关系。Rose 中使用带空心箭头的线段表示各模型元素之间的泛化关系。

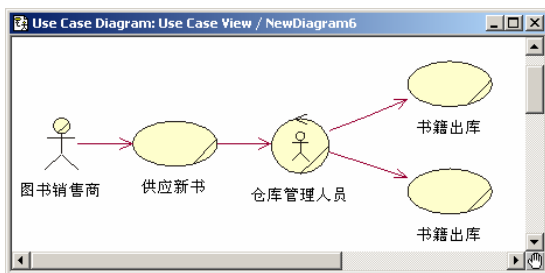


图 3-7 业务用例图中的关联关系

3.3.3 用例图

用例图 (Use Case Diagram) 用于对软件系统进行需求分析,即用于描述一个软件系统需要完成什么样的功能。用例图中的信息包括系统中的执行者和用例的描述以及两者之间的相互关系的描述。

在用例图中,主要包括用例和执行者两类模型元素。其中,用例表示软件系统中的功能模块,而执行者表示与所创建的系统进行交互的人或物。

在用例图中,模型元素之间可以建立以下 4 种关系。

(1) 关联关系描述执行者和用例之间的交互关系。例如,对于图书销售管理系统,“客户”执行者需要与“网上购书”用例进行交互,而“网上购书”用例又需要与“物流系统”执行者进行交互。在 Rose 中使用单向箭头图标来表示模型元素彼此之间的关联关系。

(2) 包含关系描述一个用例需要利用另一个用例提供的功能,本质上是一种使用关系。例如,对于图书销售管理系统中的“网上购书”用例,就需要使用“信用卡验证”用例和“网络结算”用例所提供的功能。在 Rose 中使用单向虚线箭头图标来表示元素彼此之间的包含关系,并标注上 <<include>>, 如图 3-8 所示。

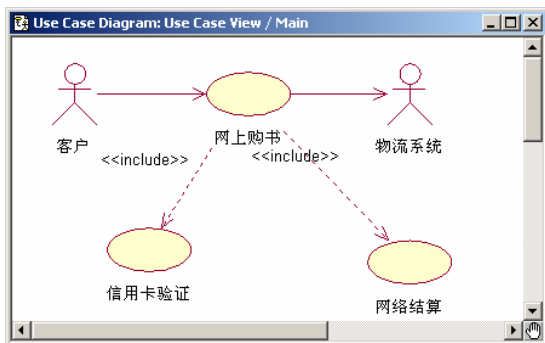


图 3-8 用例图中的包含关系

(3) 扩展关系描述一个用例对另外一个用例的功能进行扩展，在原有功能的基础上增加一些新的功能。例如，对于图书销售管理系统，它的“网络结算”用例描述正常的结算工作，但是如果某个客户的信用卡是在异地银行办理的，那么就不能进行普通的“网络结算”，此时可以通过扩展关系，在“异地银行结算”用例中处理这种情况。在 Rose 中使用单向虚线箭头图标来表示元素彼此之间的扩展关系，并标注<<extend>>，如图 3-9 所示。

(4) 泛化关系描述执行者之间或用例之间的抽象与具体、一般与特殊的关系。例如，对于网络购书的客户可以将其分为两类，一类是网络书店的会员客户，另一类是普通客户。对于这两类客户可能需要有不同的销售价格。如图 3-10 所示，客户是抽象的执行者，会员客户和普通客户是更为具体的执行者，他们之间存在着泛化关系。

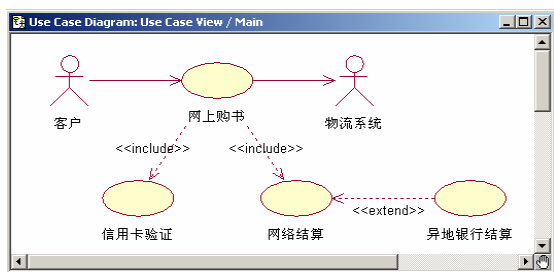


图 3-9 用例图中的扩展关系

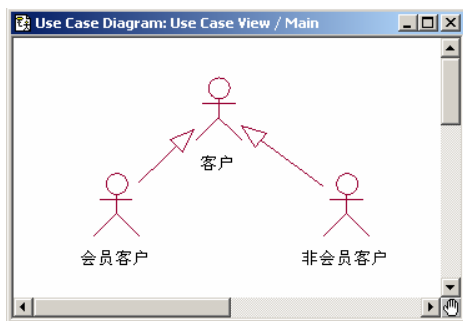



图 3-10 用例图中的泛化关系

3.3.4 类图

类是面向对象的软件开发中的一个核心概念。Rose 中的类图用于描述软件系统中涉及的类的相关信息，以及类与类之间的相互关系。类的信息包括类的属性和操作，类图需要描述类属性的名称、类型，类操作的名称、参数列表、返回类型，以及类与类之间的聚集、泛化、依赖和泛化关系。

类图创建在浏览窗口的逻辑视图（Logic View）下，并且逻辑视图中一般已有一个自动创建的名为 Main 的类图。通过双击浏览窗口中的类图图标  Main 可以打开该类图；也可以通过右键快捷菜单选择菜单项“New►Class Diagram”创建新的类图。

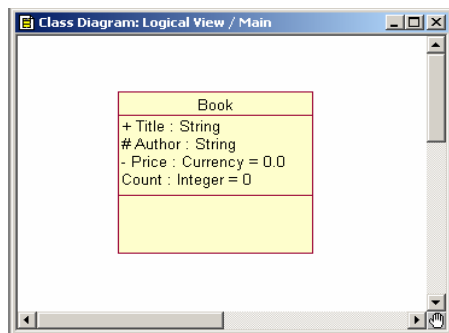



图 3-11 Book 类的属性设置



类图中最主要的模型元素就是类，通过选择类图右侧工具栏中的“类”（Class）按钮 ，可以在类图中创建一个新的类。类创建以后，首先必须为其指定一个类名，接着需要为其增加相应的属性和行为。

类属性的设置包括设置该属性的名称、数据类型、初始值、访问控制属性等相关规范。如图 3-11 所示，一个类名为 Book 的类拥有 4 个属性 Title、Author、Price、Count，它们有不同的可见性、数据类型和初始值。

类除了拥有属性以外，还可以为其定义相应的操作或者行为。为类定义操作需要设置该操作的名称、参数表和返回值的数据类型等相关规范。其格式如下：

操作名称（参数 1：数据类型，参数 2：数据类型……）：操作的返回数据类型

在一个类图中的多个类之间还可能存在着彼此的相互关系，Rose 中可以在类之间定义关联、聚集、泛化、依赖关系。

(1) 关联关系是类与类之间的一种词法连接，使得一个类可以访问或使用另一个类的公共属性和操作，实现在不同类之间的交互和通信。关联关系又分为单向关联和双向关联，分别使用  按钮和  按钮创建。

(2) 依赖关系表示一个类需要引用另一个类的定义，其目的也是为了实现不同类之间的信息交换。但依赖关系与关联关系不同之处在于：它们对程序代码有不同的影响；依赖关系只能是单向的，而关联关系可以是单向的也可以是双向的；依赖关系使用虚线箭头表示。

(3) 聚集关系表示的是类之间“整体与部分”的关系。如图 3-12 所示，该图描述了“Sale List”（订单列表）类和“Sale Order”（订单）类之间的聚集关系。

要注意的是，箭头的方向是由表示“整体”的类指向表示“部分”的类的。

(4) 泛化关系表示类之间“一般与特殊”、“抽象与具体”的关系，即不同类之间的继承关系。如图 3-13 所示，该图表示了“Sale Order”（订单）类与“Wholesale Sale Order”（批发订单）类、“Retail Sale Order”（零售订单）类之间的泛化关系。

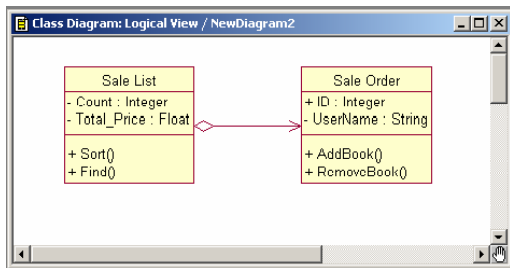


图 3-12 类之间的聚集关系

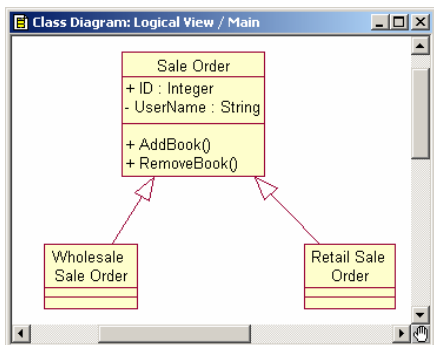


图 3-13 类之间的泛化关系

需要注意的是，箭头的方向是由表示“具体”或“特殊”的类指向表示“一般”或“抽象”的类的。

3.3.5 协作图与时序图

在 Rose 中，表示模型系统中对象之间的交互行为的图有两种：协作图和时序图。其中，协作图按照对象本身进行组织，展示了对象之间的连接，以及连接的对象之间如何发送/接收消息。而时序图则用来描述对象之间动态的交互行为，着重体现对象间消息传递的时间顺序。

选择菜单项“New►Collaboration Diagram”，在 Rose 的逻辑视图（Logic View）中创建



一个协作图以后，接着就是在其中增加对象，同时为新增的对象设置规范，包括设置该对象的名称、对应的类名、说明文档等内容。

为了描述系统中对象之间的交互关系，在协作图中的对象被设置以后，需要在对象之间建立链接。对象之间的链接用实线表示。除了可以在不同的对象之间建立链接外，在同一个对象之上也可以建立特殊的“反身链接”（Link to Self）。

链接建立以后，需要在链接上添加消息，表示对象之间传送的信息的内容。消息的类型可以是“链接消息”（Link Message）或“反向链接消息”（Reverse Link Message）。

最后，Roes 还允许开发人员将对象之间传送的消息映射为对象的操作。需要注意的是，消息是被映射为接收该消息对象的操作，而不是发送该消息对象的操作。

在图 3-14 中，可以看到一个协作图的例子，其中有两个对象之间的链接和在它们之间传送的 3 个消息，以及把消息映射为对象的操作。

选择菜单项“New ► Sequence Diagram”，可以在 Rose 的逻辑视图（Logic View）中创建时序图。打开时序图以后，可以在其中增加对象，并为该对象设置对象所在的类、对象的持续性等属性。对象设置以后，按照时间顺序，在时序图中从上到下，依次在对象之间添加表示消息传送的箭头符号。与协作图相同，在时序图中同样可以为对象之间的消息指定其映射的对象操作。方法是，在消息上右击，从弹出菜单中选择该消息所映射的操作或者输入一个新操作。

图 3-15 中显示了在时序图中如何表示两个对象 Object1 和 Object2 之间的交互行为。

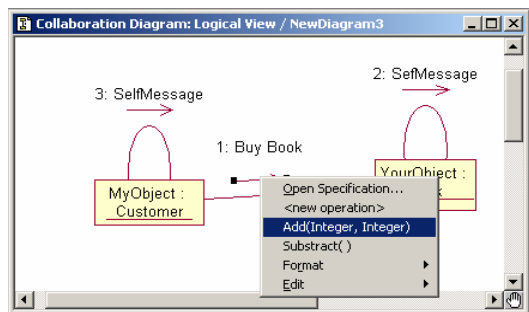


图 3-14 协作图例子

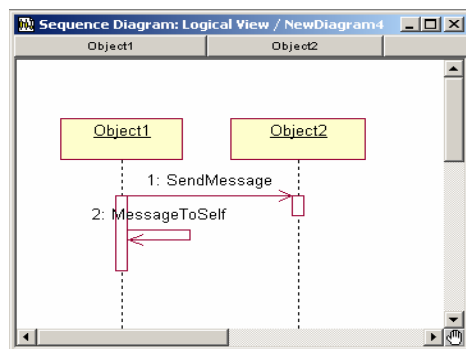


图 3-15 时序图

3.3.6 活动图

活动图通常用于建模用例的事件流，描述一个操作完成所需要的活动步骤。活动图根据对象状态的变化来获取活动和它们的结果，它表示了各个活动及其之间的关系。

在浏览窗口中选择菜单项“New ► Activity Diagram”，可以创建新的活动图。在活动图中首先需要增加“泳道”（SwimLane），并在其顶部为其命名。接着需要在相应的“泳道”中添加开始状态和结束状态；然后根据用例的事件流添加相应的活动；同时还要在活动之间设置转换和转换发生需要具备的条件。

在图 3-16 的例子中可以看到“订单处理”操作的活动图：

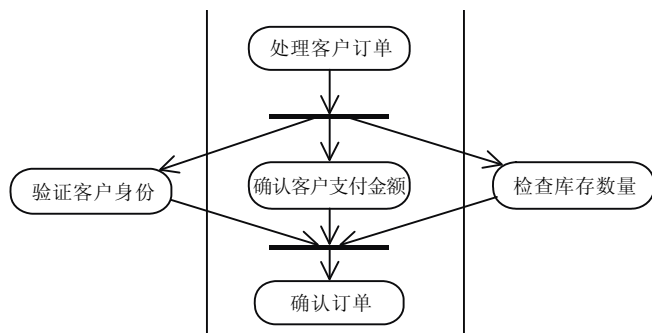


图 3-16 活动图

3.3.7 状态图

一个状态图用于描述一个类的实例（对象）在其生命期中所处的不同状态，以及对象在不同状态之间进行的转换和进行这些转换的条件。这些信息可以用于类的详细设计，开发人员可以使用类的状态信息设计和编制类。

在浏览窗口中选择菜单项“New ► Statechart Diagram”，可以创建新的状态图。在状态图中可以加入对象的各种不同状态，其中包括两种特殊的状态：“初始状态”和“结束状态”。在图 3-17 的例子中可以看到在图书销售管理系统中，Book 类对象的各种不同状态。

在状态图中，还可以对状态的各种规范和相关属性进行增加和设置，其中包括该状态的名称、原型和说明文档以及与该状态有关的活动。这里的活动是指对象在特定状态时的行为。

状态设置完成以后，需要在状态之间增加彼此的转换和设置与转换有关的属性。状态之间的转换使用带箭头的线段表示，箭头的方向由转换之前的状态指向转换之后的状态。也可以为某个状态增加一个到自身的状态转换，称之为“反身转换”。对新增加的状态转换，可以为其设置属性和规范。其中包括：导致状态转换的事件名称、原型、相关参数、该转换的说明文档、该状态转换发生时需要满足的条件、该状态转换进行时发生的动作、转换的初始状态和结束状态等信息。

在图 3-18 的例子中可以看到在图书销售管理系统中，Book 类对象的完整状态图。

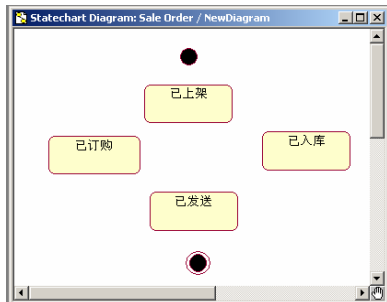


图 3-17 增加状态

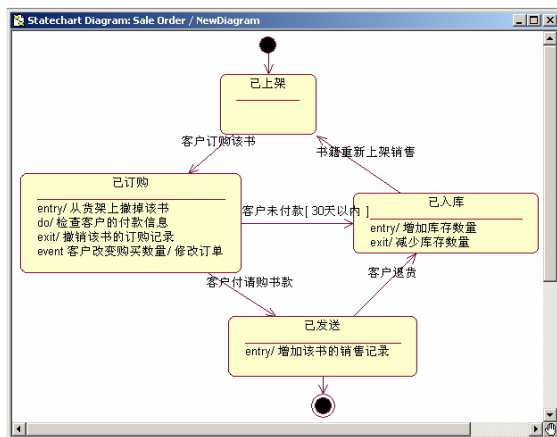


图 3-18 状态图



3.3.8 构件图和部署图

1. 构件图

构件图用于描述组成软件系统的各个构件之间的依赖关系。构件是代码的物理模块，主要包括：源代码文件、二进制目标文件和可执行文件等。

在浏览窗口中的“Component View”项目上右击，选择菜单项“New► Component Diagram”，可以创建新的构件图。构件图由构件和构件之间依赖关系组成。

在增加新的构件时，可以为该构件设置相应的规范，包括构件的原型、构件使用的语言、构件的说明文档等构件的配置属性。

构件之间的依赖关系是指构件之间在编译、链接或执行时的相互关系。例如，如果构件 A 依赖构件 B，则意味着构件 A 需要使用构件 B 提供的功能，那么构件 B 需要在构件 A 之前被编译，并且当构件 B 被修改以后，构件 A 需要重新编译。

例如，在图 3-19 所示的构件图中描述了主程序构件与书籍管理构件、订单管理构件和账户管理构件之间的依赖关系。

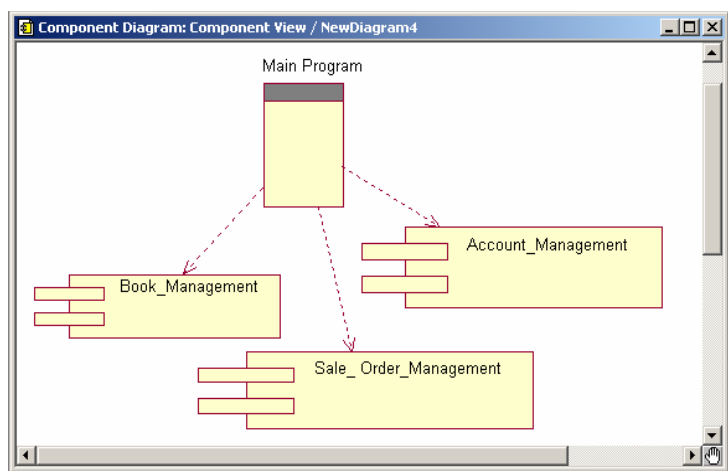


图 3-19 构件图

2. 部署图

部署图用于描述在软件系统运行时进行计算处理的结点和在结点上活动的构件的配置情况，它描述了处理器、设备和软件构件运行时的体系结构。

部署图由结点和结点之间的连接组成。部署图中的结点主要包括处理器和设备两类。处理器是具有独立数据处理能力的智能机器。设备是指没有处理能力的硬件设备。部署图描述这些机器或设备之间的物理连接。

图 3-20 所示为一个图书销售管理系统的部署图实例。

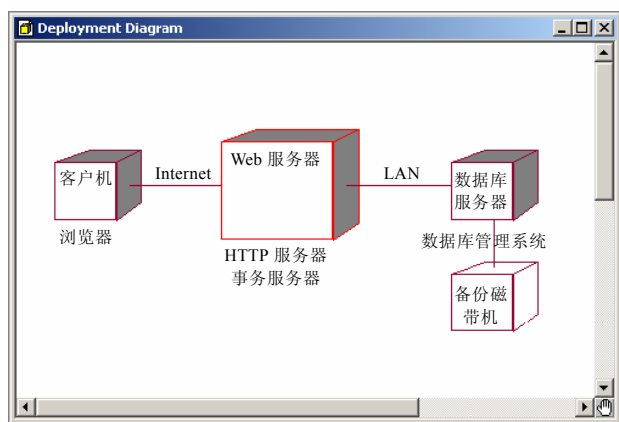


图 3-20 部署图

习题 3

- 3.1 什么是软件开发工具？按照功能进行分类，可以将软件开发工具分成哪几大类？
- 3.2 列出常用的软件开发工具，并按照本章的分类标准对其进行分类。
- 3.3 集成化的 CASE 环境相对于彼此独立的软件开发工具有哪些明显的优势？
- 3.4 软件开发工具的集成可以分成哪几个层次？
- 3.5 Rational Rose 可以建立哪几种模型图？
- 3.6 简述业务用例图与用例图的区别？
- 3.7 协作图和时序图都用于描述模型系统中对象之间的交互行为，请简述它们各自的特点。
- 3.8 在 Rose 的类图中，类之间存在 4 种关系：关联、聚集、泛化和依赖，请简述它们的不同之处。
- 3.9 请使用 Rose 绘制“网上商店购物管理系统”中“购物”流程的活动图。
- 3.10 请使用 Rose 绘制“自动取款机 ATM (Auto Trade Machine) 系统”的用例图。

第 4 章 面向对象方法

面向对象的开发方法（Object-Oriented Software Development，OOSP）是一种新的软件工程方法，其基本思想是尽可能地按照人类认识世界的方法和思维方式来分析和解决问题，这种方法能够提供更加清晰的需求分析和设计，是指导软件开发活动的系统方法。

但是在面向对象的领域，目前大多数工作都集中在编程语言上。面向对象的开发方法，贯穿了整个软件生命期，其中面向对象的分析与设计是面向对象开发的关键，因此，本章主要讨论面向对象的需求分析（OOA）与面向对象设计（OOD）的基本概念。



4.1 面向对象方法概述

面向对象的思想最初出现于挪威斯陆大学和挪威计算机中心共同研制的 Simula 67 语言中，其后，随着位于美国加利福尼亚州的 Xerox 研究中心推出 Smalltalk-76 和 Smalltalk-80 语言，面向对象的程序设计技术开始迅猛发展。面向对象的概念和应用已超越了程序设计和软件开发，扩展到很宽的范围，如数据库系统、交互式界面、分布式系统、网络管理结构和人工智能等领域。一些新的工程概念及其实现，如并发工程、综合集成工程等也需要面向对象的支持，所以面向对象的方法已成为当今软件开发的主流方法。



4.1.1 什么是面向对象方法

自 20 世纪 70 年代末以来，传统的软件工程方法对克服“软件危机”，促进软件产业的发展起到了重要作用，自顶而下的分析和设计方法、软件项目的工程化管理、软件工具和开发环境及软件质量保障体系都有力地推动了软件能力的解决。

但随着软件形式化方法及新型软件的开发，传统的软件工程方法的局限性逐渐暴露出来，存在的主要问题如下。

（1）传统软件开发方法无法实现从问题空间到解空间的直接映射

传统软件开发方法求解过程是，先对应用领域（问题空间）进行分析，建立起问题空间的逻辑模型，再通过一系列复杂的转换和算法，构造计算机系统，获得解空间。由于问题空间与解空间的模型、描述方式的不同，它们之间存在着复杂的转换过程，尤其对于复杂系统及普遍存在的需求变化，更难以适应。图 4-1 描述了它的转换过程。



图 4-1 传统的软件开发方法的映射过程

(2) 传统软件开发方法无法实现高效的软件复用

由于软件系统本质上是信息处理系统,传统的软件工程方法是面向过程的,将数据和处理过程(操作)分离,不仅增加了软件开发的难度,也难于支持软件复用。

(3) 传统软件开发方法难以实现从分析到设计的直接过渡

分析和设计是软件开发的两个最重要的阶段。传统的软件开发方法在这两个阶段所建立的模型通常是完全不同的,例如,结构化方法,主要的分析模型是分层的DFD图,而设计阶段的模型则是软件结构图(SC),从分析模型到设计模型要经过复杂的变换过程。

面向对象的方法则是将软件系统看做一系列离散的解空间对象的集合,并使问题空间的对象与解空间对象尽量一致,如图4-2所示。这些解空间对象之间通过发送消息而相互作用,从而获得问题空间的解。而且,问题空间与解空间的结构、描述的模型十分一致,减少了软件系统开发的复杂度,使系统易于理解和维护。

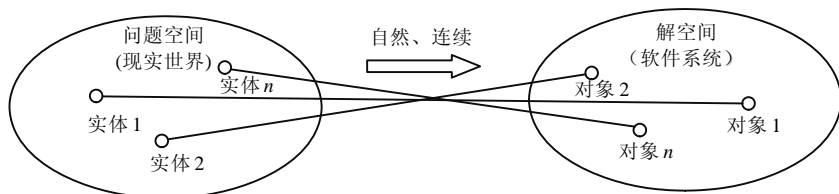


图 4-2 面向对象方法的直接转换

究竟什么是面向对象的系统?根据Coad和Yourdon的定义,如果是按照以下4个概念设计和实现的系统,则称为是面向对象的:

面向对象=对象+类+继承+通信

4.1.2 面向对象方法的主要特点

面向对象的方法具有以下主要特点。

(1) 按照人类习惯的思维方法,对软件开发过程所有阶段进行综合考虑

传统的程序设计技术是面向过程的设计方法,以算法为核心,将数据和过程作为相互独立的部分,程序代码用于处理这些数据。这种数据和代码分离的结构,反映了计算机的观点,但却忽视了数据和操作之间的内在联系,所以用这种方法设计的软件系统,其解空间与问题空间不一致。

而面向对象的方法以对象为核心,强调模拟现实世界中的概念而不是算法,尽量用符合人类认识世界的思维方式来渐进地分析、解决问题,使问题空间与解空间具有一致性,便于对软件开发过程所有阶段进行综合考虑,能有效地降低软件开发的复杂度,提高软件质量。

(2) 软件生存期各阶段所使用的方法、技术具有高度的连续性

传统的软件开发过程用瀑布模型描述,其主要缺点是把软件开发这样一个充满回溯的过程硬性分割为几个阶段,而且各个阶段所使用的模型、描述方法不相同。

而面向对象的方法使用喷泉模型作为其工作模型,软件生存期各阶段没有明显的界限,开发过程回溯重叠,使用相同的描述方法和模型,使得软件生存期各阶段所使用的方法、技



术具有高度的连续性。

(3) 软件开发各个阶段有机集成, 有利于系统的稳定性

将 OOA(Object-Oriented Analysis)、OOD(Object-Oriented Design)、OOP(Object-Oriented Program) 有机地集成在一起, 使开发过程始终围绕着建立问题领域的对象(类)模型进行, 而各阶段解决的问题又各有侧重。由于是以对象为中心构造软件系统, 而不是基于对系统功能进行分解来构造系统, 当系统功能需求改变时不会引起系统结构的变化, 使软件系统具有好的稳定性和可适应性。

(4) 具有良好的重用性

面向对象的技术在利用可重用的软件成分构造新软件系统上具有很大的灵活性, 由于对象所具有的封装性和信息隐蔽, 使得对象的内部实现与外界隔离, 具有较强的独立性, 因此, 对象类提供了较理想的可重用的软件成分。而对象类的继承机制使得面向对象的技术实现可重用性更加方便、自然和准确。



4.2 面向对象的基本概念

本节讨论几个重要的面向对象的基本概念, 这对理解面向对象的思想, 学习和掌握面向对象的开发方法是十分重要的。



4.2.1 对象与类

面向对象的方法以对象作为最基本的元素, 对象是分析问题、解决问题的核心。对象与类是讨论面向对象方法的最基本最重要的概念。

1. 对象(Object)

对象是客观事物或概念的抽象表述, 对象不仅能表示具体的实体, 也能表示抽象的规则、计划或事件。通常有以下的对象类型。

- ① 有形的实体: 在现实世界中, 每个实体都是对象, 如飞机、车辆、计算机、桌子、房子、机器等, 都属于有形的实体, 这是容易识别的对象。
- ② 作用: 指人或组织, 如教师、学生、医生、政府机关、公司、部门等, 所起的作用。
- ③ 事件: 指在某个特定时间内所发生的事, 如学习、演出、开会、办公、事故等。
- ④ 性能说明: 对产品的性能指标的说明, 例如, 计算机的配置有 CPU、硬盘及主板的速度、型号、性能说明等。

对象不仅能表示结构化的数据, 而且也能表示抽象的事件、规则以及复杂的工程实体, 这是结构化方法所不能做到的, 因此, 对象具有很强的表达能力和描述功能。

每个对象都存在一定的状态(State), 内部标识(Identity), 可以给对象定义一组操作(Operation), 对象通过其运算所展示的特定行为称为对象行为(Behavior), 对象本身的性质称为属性(Attribute), 对象将它自身的属性及运算“包装起来”, 称为“封装”(Encapsulation)。

在面向对象的系统中，对象是一个封装数据属性和操作行为的实体。数据描述了对象的状态，操作可操纵私有数据，改变对象的状态。当其他对象向该对象发出消息，该对象响应时，其操作才得以实现，在对象内的操作通常叫做方法。

2. 类（Class）

类又称对象类（Object Class）是指一组具有相同属性和运算的对象的抽象，一组具有相同数据结构和相同操作的对象的集合，类是对象的模板。在一个类中，每个对象都是类的实例（Instance），它们都可以使用类中提供的函数。例如，小轿车是一个类，红旗牌小轿车，东风牌小轿车都是它的一个对象。类具有属性，用数据结构来描述类的属性。类具有操作，它是对象行为的抽象，用操作名和实现该操作的方法（Method），即操作实现的过程来描述。

3. 对象和类的描述

类和对象的描述分别如图 4-3 和图 4-4 所示。在图 4-3 中，“人”是类名，包含两个属性：姓名和年龄，具有两种操作：改变工作和改变地址。这里“人”只是一个抽象的概念，并不代表某个具体的人。图 4-4 则描述了“人”这个类的两个对象实例。

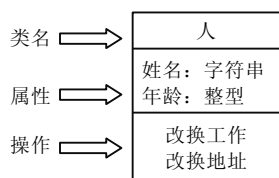


图 4-3 类的描述

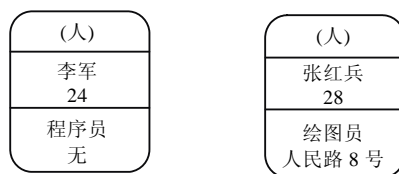


图 4-4 对象的描述

由于对象是类的实例，在进行系统分析和设计时，通常把注意力集中在类上，而不是具体的对象上。

4.2.2 继承

继承（Inheritance）是使用现存的定义作为基础，建立新定义的技术，是父类和子类之间共享数据结构和方法的机制，这是类之间的一种关系。在定义和实现一个类的时候，可以在一个已经存在的类的基础上进行，把这个已经存在的类所定义的内容作为自己的内容，并加入若干新内容。继承性通常表示父类与子类的关系（见图 4-5）。子类继承了父类的特性。继承性分为单重继承和多重继承。

单重继承：一个子类只有一个父类，即子类只继承一个父类的数据结构和方法。

多重继承：一个子类可有多个父类，继承多个父类的数据结构和方法

图 4-6 所示为继承性的一种图示描述方法。通过继承关系还可以构成层次关系，单重继承构成的类之间的层次关系是一棵树，多重继承构成的类之间的关系是一个网格（如果将所有无子类的类都看成还有一个公共子类的话）。而且继承关系是可传递的。

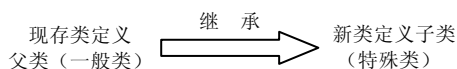


图 4-5 继承性

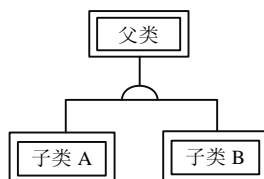


图 4-6 继承性描述

4.2.3 多态性

多态性 (Polymorphism) 是指相同的操作或函数、过程作用于不同的对象上并获得不同的结果的现象。当相同操作的消息发送给不同的对象时，每个对象将根据自己所属类中所定义的操作去执行，从而产生不同的结果。一个具有多态性的行为在外界看来，不同对象都用相同的名字去执行它，因此外界认为那些都是种行为，并预期得到相同的结果，但实际上，该行为的多态性决定了对象将根据自己所接收到消息的相关参数去选择执行该行为的哪一个版本，从而得到不同的结果。

例如，在父类“几何图形”中，定义了一个操作，它的子类“椭圆”和“矩形”都继承了几何图形的绘图操作。同是“绘图”操作，分别作用在“椭圆”和“矩形”上，却画出不同的图形。

多态性允许每个对象以适合自身的方式去响应共同的消息，这样就增强了操作的透明性、可理解性和可维护性。与多态性密切相关的有以下两个概念：

- ◎ 重载 (Overload)，是指对特殊类中继承来的属性或者操作进行重新定义；
- ◎ 动态绑定 (Dynamic Binding)，是指在运行时根据对象所接收的消息，动态地确定要连接哪一段服务代码。

4.2.4 消息

消息 (Message) 是指对象之间在交互中所传送的通信信息。一个消息应该包含以下信息：消息名、接收消息对象的标识、服务标识、消息和方法、输入信息、回答信息等。消息使对象之间互相联系、协同工作，实现系统的各种服务。

通常，一个对象向另一个对象发送信息请求某项服务，接收对象响应该消息，激发所要求的服务操作，并将操作结果返回给请求服务的对象，这种通信机制叫做消息传递。发送消息的对象不需要知道接收消息的对象如何对请求予以响应。



4.3 面向对象的分析

面向对象的分析 (Object-Oriented Analysis, OOA)，即运用面向对象的方法进行需求分析。OOA 是面向对象方法从编程领域向分析领域延伸的产物，充分体现了面向对象的概念与原则。

4.3.1 需求分析中的问题

在软件开发过程中，建立了各种分析方法，其中最具有影响的有：功能分析方法、结构化分析方法、信息建模法等。各种方法从不同的角度、不同的观点对问题域进行分析并建立系统的模型。无论使用哪种方法，需求过程都面临着以下需要解决的问题。

1. 明确问题域和系统责任的困难

问题域（Problem Domain）是指被开发系统的应用领域，即拟建立系统进行处理的业务范围。系统责任（System Responsibilities）是指所开发系统应该具备的职能。例如，银行的业务处理系统，其问题域，即“银行”，包括银行的组织机构、人事管理、日常业务等，而系统责任则包括银行的日常业务（如金融业务、个人储蓄、国债发行、投资管理等），用户权限管理、信息的定期备份等。要明确问题域和系统责任，即要获取和确定需求是困难的。

2. 充分交流的问题

在软件开发过程中，各类人员的充分交流是获得准确分析结果的关键，其中以软件开发人员与领域专家之间的交流尤为重要。由于软件开发人员大多不了解应用领域的问题，但是分析工作却要求他们在较短的时间内掌握问题域的基本情况和关键问题，而应用领域的专家多半不熟悉软件开发，所以在分析过程中，软件开发人员必须与领域专家密切配合，充分交流，才能获得对问题的准确分析。

3. 需求的不断变化

在分析过程中的另一个头痛的问题是需求总是在不断地变化。例如，用户会不断提出新的需求，经费可能会增加或减少，技术支持的缺乏和增加也会引起需求的调整。需求的变化要求分析员去修改分析，甚至重新作分析，而反复的修补常常会将系统搞乱，还可能会引入新的错误。

需求的变化是分析将面临的一个严峻问题，应变能力的强弱是衡量一种分析方法优劣的重要标准。

4. 考虑复用要求

软件复用是提高软件开发效率，改善软件质量的重要途径。软件复用的范围已经从 20 世纪 80 年代的程序复用，转移到了分析结果和设计结果的复用上，这将产生更加显著的效果。

分析结果的复用是指把分析模型中的成分组成可复用的构件，用于构造新系统进行分析时复用。为此必须解决可复用构件的提取、制作与检索，可复用构件库的组织，可复用构件的组装等问题。要求分析结果中的基本成分具有较强的独立性。为了在检索中能够有效地搜索和理解构件，要求分析结果中的可复用成分与问题域中的事物具有良好的对应关系。

对于以上软件需求中所面临的主要问题，尤其是需求不断变化的问题和软件复用的问题，传统的软件开发方法由于本身的局限性，已不可能找到有效的解决方案。



4.3.2 OOA的特点

一个好的分析方法，应该能够有效地解决上述软件需求中的问题。OOA 在解决这些问题上有较强的能力。

1. 有利于对问题及系统责任的理解

OOA 强调从问题域中的实际事物及与系统责任有关的概念出发来构造系统模型。系统中对象及对象之间的联系都能够直接地描述问题域和系统责任，构成系统的对象和类都与问题域有良好的对应关系，因此十分有利于对问题及系统责任的理解。

2. 有利于对人员之间的交流

由于 OOA 与问题域具有一致的概念和术语，同时尽可能使用符合人类的思维方式来认识和描述问题域，因此使软件开发人员和应用领域的专家具有共同的思维方式，理解相同的术语和概念，从而为他们之间的交流创造了基本条件。

3. 对需求变化有较强的适应性

在一般系统中，最容易变化的是功能（在 OO 方法中是操作），其次是与外部系统或设备的接口部分，再就是描述问题域中事物的数据。系统中最稳定的部分是对象。

为了适应需求的不断变化，要求分析方法将系统中最容易变化的因素隔离起来，并尽可能减少各单元之间的接口。

在 OOA 中，对象是构成系统最基本的元素，而对象的基本特征是封装性，将容易变化的成分（如操作及属性）封装在对象中，这样对象的稳定性使系统具有宏观上的稳定性。即使需要增减对象时，其他的对象也具有相对的稳定性。因此，OOA 对需求的变化具有较强的适应性。

4. 支持软件复用

OO 方法的继承性本身就是一种支持复用的机制，子类的属性及操作不必重新定义，可由父类继承而得。无论在分析、设计阶段还是在编码阶段，继承性对复用都起着极其重要的作用。

OOA 中的类也很适合作为可复用的构件，因为类具有完整性，它能够描述问题域中的一个事物，包括其数据和行为的特征；类具有独立性，是一个独立的封装的实体。完整性和独立性是实现软件复用的重要条件。



4.3.3 OOA的基本任务与分析过程

1. OOA的基本任务

OOA 是软件开发过程中的问题定义阶段，其目标是完成对所求解问题的分析，确定系统“做什么”，并建立系统的分析模型。

运用面向对象的方法，对问题域和系统责任进行分析和理解，找出描述它们的类和对象，

定义其属性和操作,以及它们的结构,包括静态联系和动态联系,最终获得一个符合用户需求,并能够反映问题域和系统责任的 OOA 模型。

通过 OOA 建立的系统模型是以对象概念为中心的,因此又称为概念模型,它由一组相关的类组成。OOA 可以采用自顶而下的方法,逐层分解建立系统模型,也可以自底而上地从已有定义的基类出发,逐步构造新类。

2. OOA的分析过程

面向对象的分析过程分为论域分析和应用分析,该阶段的目标是获得对问题论域的清晰、精确的定义,产生描述系统功能和问题论域的基本特征的综合文档。

(1) 论域分析 (Domain Analysis)

论域分析过程是抽取和整理用户需求并建立问题域精确模型的过程。其主要任务是充分理解专业领域的业务问题和投资者及用户的需求,提出高层次的问题解决方案。

应具体分析应用领域的业务范围、业务规则和业务处理过程,确定系统范围、功能、性能,完善细化用户需求,抽象出目标系统的本质属性,建立问题论域模型。

(2) 应用分析 (Application Analysis)

应用分析是指将论域分析建立起来的问题论域模型,用某种基于计算机系统的语言表示出来。响应时间需求、用户界面需求和数据安全等特殊的需求都在这一层分解抽出。图 4-7 描述了 OOA 分析过程的具体步骤。

① 获取用户基本需求

首先,用户与开发者之间进行充分交流,通常使用用例 (User Case) 来收集和描述用户的需求,然后标识使用该系统的不同的行为者 (Actor),所提出的每个使用场景 (或功能) 称为一个用例。建立的系统所有用例,则构成完整的系统需求。

② 标识类和对象

标识类与对象是一致的。在确定系统的用例后,可标识类及类的属性和操作。从问题域或用例的描述入手,发现类及对象。列出对象可能有的形式:外部实体、事物、发生的事件、角色、组织单位、场所、构造物等。

在此基础上,进一步确定最终对象。通常可根据以下原则确定:保留对象具有需要保留的信息,需要的服务,具有多个属性,具有公共属性及操作。标识类 (对象) 属性,从本质上讲,属性定义了类,可从问题的陈述中或通过类的理解而标识出属性。定义操作,操作定义了对象的行为并以某种方式修改对象的属性。操作分为:对数据的操作、计算操作和控制操作。

③ 定义类的结构和层次

类的结构有:一般与特殊 (Generalization-Specialization) 结构,整体与部分 (Whole-Part) 结构。构成类图的元素所表达的模型信息,通常分为三个层次,如图 4-8 所示。

◎ 对象层,给出系统中所有反映问题域和系统责任的对象。

◎ 特征层,给出类 (对象) 的内部特征,即类的属性和操作。

◎ 关系层,给出各类 (对象) 之间的关系,包括继承、组合、一般—特殊、整体—部分、属性的静态依赖关系,操作的动态依赖关系等。

④ 建立类 (对象) 之间的关系用“对象—关系模型”描述系统的静态结构。

建立“对象—行为”模型,描述系统的动态行为。

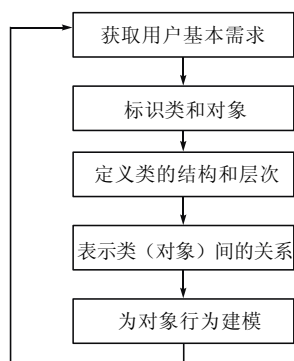


图 4-7 OOA 分析过程

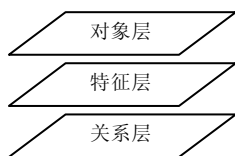


图 4-8 OOA 基本模型



4.4 面向对象的设计

面向对象的设计（Object-Oriented Design, OOD）是面向对象方法（OO）的核心阶段。按照描述 OO 方法的“喷泉模型”，软件生命期的各阶段交叠回溯，整个生命期的概念、术语、描述方式具有一致性，因此从分析到设计无须表示方式的转换，只是分析和设计的任务分工与侧重不同。

OOA 建立的是应用领域面向对象的模型，而 OOD 建立的则是软件系统的模型。与 OOA 的模型比较，OOD 模型的抽象层次较低，因为它包含了与具体实现有关的细节，但是建模的原则和方法是相同的。

4.4.1 OOD的准则

建立 OOD 模型，可以看做是按照设计的准则，对分析模型进行细化。虽然这些设计准则并非面向对象的系统独用，但对面向对象设计起着重要的支持作用。面向对象的设计准则如下。

1. 抽象

抽象是指强调实体的本质、内在的属性，而忽略了一些无关紧要的属性。在系统开发中，分析阶段使用抽象仅仅涉及应用域的概念，在理解问题域以前不考虑设计与实现。而在面向对象的设计阶段，抽象概念不仅用于子系统，在对象设计中，由于对象具有极强的抽象表达能力，而类实现了对象的数据和行为的抽象。

2. 信息隐蔽

信息隐蔽在面向对象的方法中的具体体现就是“封装性”，封装性是保证软件部件具有优良的模块性的基础。封装性是指将对象的属性及操作（服务）结合为一个整体，尽可能屏蔽对象的内部细节，软件部件外部对内部的访问通过接口实现。

类是封装良好的部件，类的定义将其说明（用户可见的外部接口）与实现（用户内部实

现) 分开, 而对其内部的实现按照具体定义的作用域提供保护。对象作为封装的基本单位, 比类的封装更加具体、更加细致。

3. 弱耦合

按照抽象与封装性, 弱耦合是指子系统之间的联系应该尽量少。子系统应具有良好的接口, 子系统通过接口与系统的其他部分联系。

4. 强内聚

强内聚是指子系统内部由一些关系密切的类构成, 除了少数的“通信类”外, 子系统内的类应该只与该子系统内的其他类协作, 构成具有强内聚性的子系统。

5. 可重用

只有构建独立性强(弱耦合、强内聚)的子系统, 才能够有效地提高所设计的部件的可重用性。

4.4.2 OOD的基本任务

面向对象的设计是面向对象方法在软件设计阶段应用与扩展的结果, 将 OOA 所创建的分析模型转换为设计模型, 解决如何做的问题。面向对象的设计主要目标是提高生产效率, 提高质量和提高可维护性。

OOA 主要考虑系统做什么, 而不关心系统如何实现的问题。在 OOD 中为了实现系统, 需要以 OOA 模型为基础, 重新定义或补充一些新的类, 或在原有类中补充或修改一些属性及操作。因此, OOD 的目标是产生一个满足用户需求的, 可实现的 OOD 模型。

面向对象的设计还可以细分为系统设计和对象设计。系统设计确定实现系统的策略和目标系统的高层结构。对象设计确定解空间中的类、关联、接口形式及实现服务的算法。系统设计与对象设计之间的界限比分析与设计之间的界限更加模糊。

1. 系统设计

系统设计的任务包括: 将分析模型中紧密相关的类划分为若干子系统(也称为主题), 子系统应该具有良好的接口, 子系统内的类相互协作。标识问题本身的并发性, 将各子系统分配给处理器, 建立子系统之间的通信。

进行系统设计的关键是子系统的划分, 子系统由它们的责任及所提供的服务来标识, 在 OOD 中, 这种服务是完成特定功能的一组操作。

将划分的子系统组织成完整的系统时, 有水平层次组织和垂直块组织两种方式。层次结构又分为封闭式和开放式。所谓封闭式是指每层子系统仅使用其直接下层的, 这就降低了各层之间的相互依赖性, 提高了易理解性和可修改性。开放式则允许各层子系统使用其下任一层的子系统提供的服务。块状组织是把软件系统垂直地划分为若干个相对独立的、弱耦合的子系统, 一个子系统(块)提供一种类型的服务。图 4-9 描述了一个典型应用系统的组织结构, 系统采用了层次与块状的混合结构。通常, OOD 模型(即求解域的对象模型)也与 OOA 模型(问题域的对象模型)一样, 由主题、属性、类与对象、结构和服务 5 个层次



组成。此外，大多数系统的 OOD 在逻辑上都由 4 部分组成，这 4 部分是组成目标系统的子系统，它们是：问题域部件、人机交互部件、任务管理部件和数据管理部件。当然，在不同的软件系统中，这 4 个部件的规模和重要性差异很大。有关各部件的设计将在 Coda/Yourdon 方法中介绍。

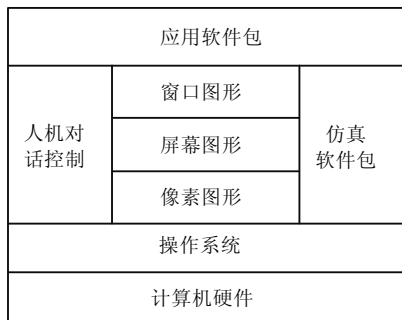


图 4-9 典型应用系统的组织结构

2. 对象设计

在面向对象的系统中，模块、数据结构及接口等都集中地体现在对象和对象层次结构中，系统开发的全过程都与对象层次结构直接相关，是面向对象系统的基础和核心。面向对象的设计通过对象的认定和对象层次结构的组织，确定解空间中应存在的对象和对象层次结构，并确定外部接口和主要的

数据结构。

对象设计是指为每个类的属性和操作进行详细设计，包括属性和操作的数据结构及实现算法，以及类之间的关联。另外，在 OOA 阶段，将一些与具体实现条件密切相关的对象，例如，与图形用户界面（GUI）、数据管理、硬件及操作系统有关的对象推迟到 OOD 阶段考虑。

在进行对象设计的同时也要进行消息设计，即设计连接类与它的协作者之间的消息规约（specification of the messages）。

3. 设计优化

对设计进行优化，主要涉及提高效率的技术和建立良好的继承关系的方法。提高效率的技术包括增加冗余关联以提高访问效率，调整查询次序，优化算法等技术。建立良好的继承关系是优化设计的重要内容，通过对继承关系的调整实现。



4.5 典型的面向对象方法

随着面向对象开发方法的发展，逐渐形成了几个主要的流派：Coad/Yourdon 方法、Booch 方法、OMT 方法和 OOSE 法。它们各有特色，主要在描述方式、图例方面考虑的重点有所不同。但在描述方式上都具有以图形方式为主的特性。



4.5.1 Booch方法

G.Booch 于 1991 年推出了 OOA&OOD 法，1994 年又发表了第 2 版。Booch 方法的开发模型包括静态模型和动态模型：静态模型分为逻辑模型和物理模型，描述了系统的构成和结构；动态模型包括状态图和时序图。

该方法对每一步都做了详细的描述，描述手段丰富、灵活。不仅建立了开发方法，还提出了设计人员的技术要求，以及不同开发阶段的人力资源配置。Booch 方法的基本模型包括类图与对象图，主张在分析和设计中既使用类图，也使用对象图。

1. 类图

类图表示系统中的类与类之间的相互关系。如图 4-10 所示，类用虚线的多边形表示。类之间的关系有关联、继承、包含和使用等。图 4-11 所示为温室管理系统的类图。

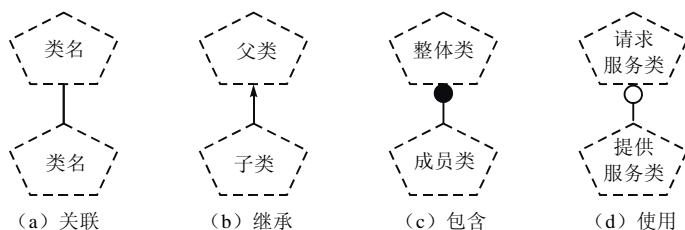


图 4-10 类图的表示

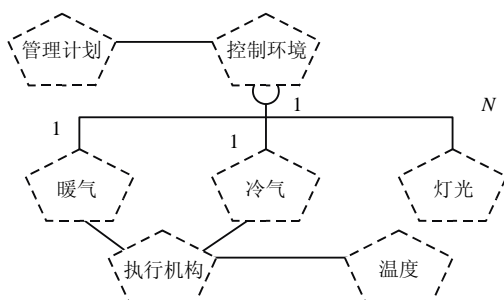


图 4-11 温室管理系统的类图

2. 对象图

对象图由对象和消息组成，如图 4-12 所示，对象用实线的多边形表示。图 4-13 描述了一个温室管理系统的对象图。

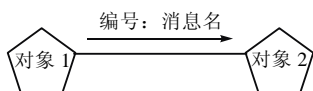


图 4-12 对象图的表示

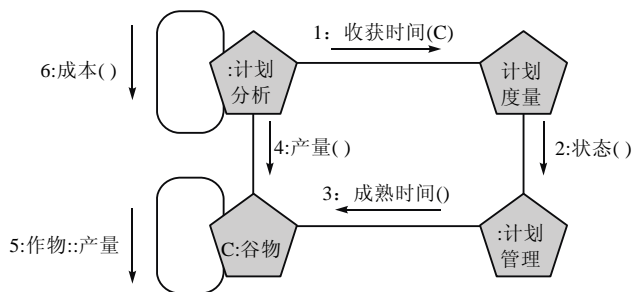


图 4-13 一个温室管理系统的对象图

3. 状态图

Booch 方法中的状态图用于描述某个类的状态空间，以及状态的改变和引起状态改变的事件，描述了系统中类的动态行为。图 4-14 给出了状态图的表示：圆角矩形表示状态，框内标注状态名；实心圆表示开始状态；状态之间的有向连线，表示引起状态改变的事件，连线上标注事件名。

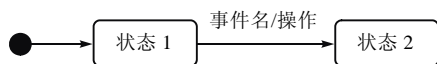


图 4-14 状态图的表示



图 4-15 所示为一个温室管理系统中环境控制器类的状态图。状态图有“空闲”、“白天”和“夜晚”三种状态。当没有种植物时，控制器处于“空闲”状态。当有种植物时，要进行温度控制，“定义气候”将根据不同季节时期，确定不同的基本温度。当处于“白天”状态时，由于温度上升，需要调节温度，使温度下降以保持温度。当处于“夜晚”状态时，也需要调节温度，使温度上升以保持温度。当种植物收割后，“终止气候”，又恢复到“空闲”状态。

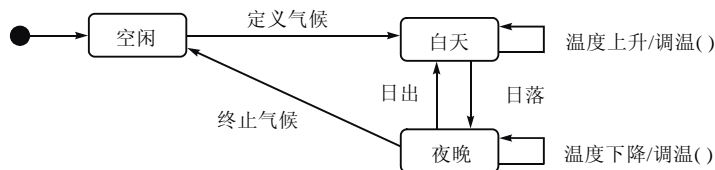


图 4-15 环境控制器类的状态图

4. 时序图

时序图用来描述对象之间交互的时间特性。时序图（见图 4-16）中，参与交互的对象放在上端，对象下的竖线，称为对象的生命线，从上到下表示时间的延伸，生命线之间带箭头的连线表示消息的传送，并在连线上标注消息名。

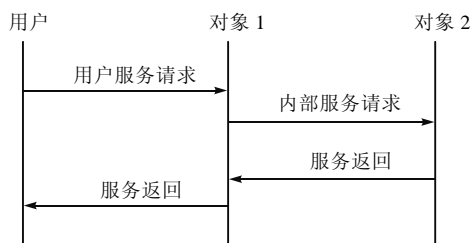


图 4-16 温室管理系统的时序图

5. 模块图

模块图表示程序构件（模块）及其构件之间的依赖关系。温室管理系统的模块图如图 4-17 所示。

6. 进程图

进程图描述系统的物理模型，在多处理器系统中，进程图描述了可同时执行的进程在各处理器上执行的情况。在单处理器系统中，进程图表示同时处于活动状态的对象。温室管理系统的进程图如图 4-18 所示。

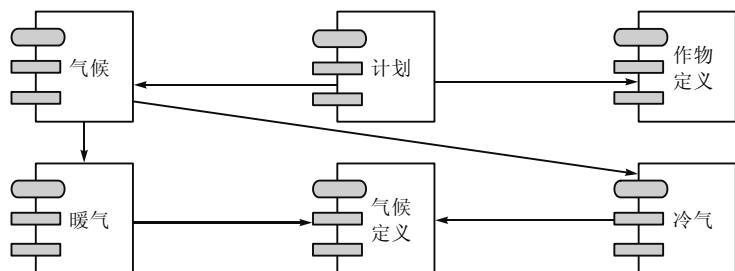


图 4-17 温室管理系统的模块图

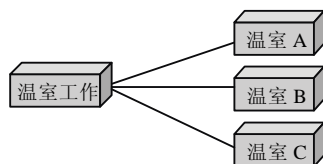


图 4-18 温室管理系统的进程图

4.5.2 Coad/Yourdon方法

Coad/Yourdon 方法由 P.Coad 和 E.Yourdon 于 1990 年推出，该方法主要由面向对象的分析（OOA）和面向对象的设计（OOD）构成，特别强调 OOA 和 OOD 采用完全一致的概念和表示法，使分析和设计之间不需要表示法的转换。该方法的特点是：表示简练、易学，对于对象、结构、服务的认定较系统、完整，可操作性强。

1. OOA

在 Coad/Yourdon 方法中, OOA 的主要任务是建立问题域的分析模型。OOA 方法分析过程和构造 OOA 概念模型的顺序由 5 个层次组成, 这 5 个层次是: 类与对象层、属性层、服务层、结构层和主题层。它们分别表示分析的不同侧面。

图 4-19 给出了每个层次中所涉及的主要概念和相应的图形表示。面向对象的分析主要有以下活动。

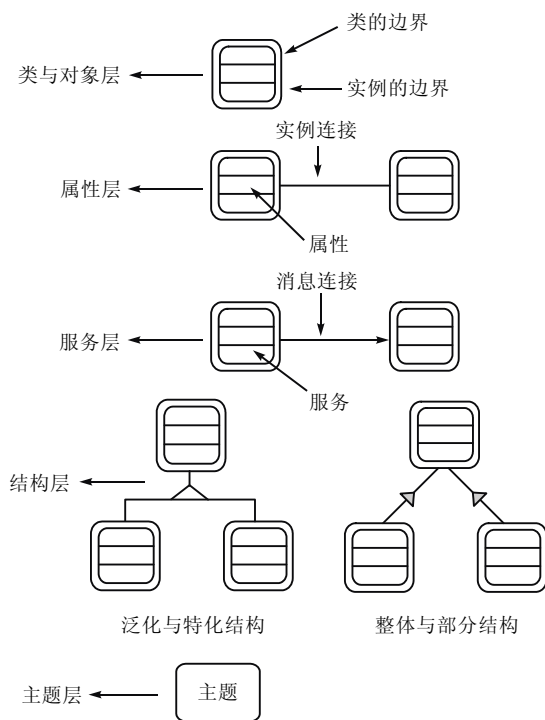


图 4-19 分析过程的 5 个层次

(1) 类和对象的认定

面向对象分析的核心是确定系统的类及对象, 它们是构成软件系统的基本元素。常用的认定方式有: 简单系统对象的认定方法和复杂系统对象的认定方法。

(2) 结构的认定

结构的认定是指描述类及对象之间的结构关系, 用来反映问题空间中复杂事物和复杂关系。有两种结构: 分类结构针对的是事物类别之间的组织关系, 组装结构对应于事物的整体与部分之间的关系。

(3) 主题的认定

主题是一种帮助理解复杂模型的抽象机制, 它将关系较密切的类及对象组织在一起, 由主题起控制作用。整个系统由若干主题构成, 便于用户从不同粒度来理解系统。

(4) 定义属性

属性是类的性质, 它是某种数据或状态信息。



(5) 定义服务

定义服务是指定义类和对象的服务和消息连接。服务是在对象接受到一条消息后所要进行的加工,它是对象表现的具体行为。而消息关联用于表示对象间的通信,说明服务的要求。通信的基本方式是消息传递。

图 4-20 和图 4-21 给出 Coad/Yourdon 方法两个应用实例。

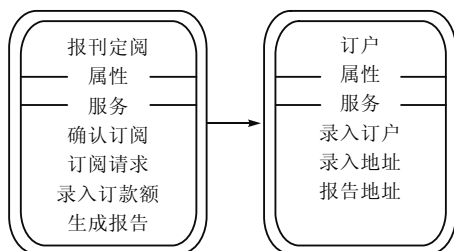


图 4-20 服务层的例子

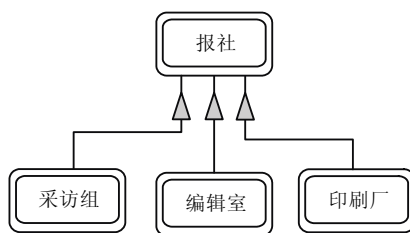


图 4-21 泛化与特化结构的例子

2. OOD

面向对象的设计通过对对象的认定和对象层次结构的组织,确定解空间中应存在的对象和对象层次结构,并确定外部接口和主要的数据结构。面向对象设计的主要目标是提高生产效率,提高质量及提高可维护性。面向对象方法中一个主要目标就是保持问题域组织框架的完整性。

OOD 的设计模型在面向对象的分析模型的 5 个层次上由 4 个部件构成,如图 4-22 所示。这 4 个部件是:问题域部件、人机交互部件、任务管理部件和数据管理部件。面向对象设计的主要内容包括设计这 4 类部件的活动。

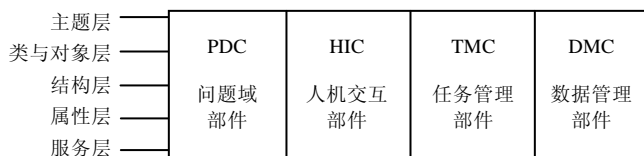


图 4-22 Coad 方法的 OOD 模型

(1) 问题域部件 (Problem Domain Component)

OOA 与具体实现无关,而 OOD 则在很大程度上受具体实现环境的约束。通过分析得到的精确模型为设计问题域部件奠定了良好的基础。通常,进行问题域部件设计只需从实现的角度出发,对通过分析所建立的问题域模型做一些修改和补充即可,例如,对类、对象、结构、属性及服务进行增加、修改或完善。设计问题域部件时可采用以下方法。

- ◎ 利用重用设计加入现有类。现有类是指面向对象的程序设计语言所提供的类库中的类,将其中所需要的类加入到问题域部件中,并指出现有类中不需要的属性及操作。
- ◎ 引入一个“根”类,将专门的问题域类组合在一起,或引入一个附加的抽象类,以便为大量的具体类定义一个相似的服务集合,建立一个协议。
- ◎ 调整继承的支持层次。如果分析模型中包括多重继承,而使用的程序设计语言中没有多重继承机制,可使用化为单一层次的方法,将多重继承化为单重继承。

(2) 人机交互部件 (Human Interaction Component)

人机交互部件用来表示用户与系统的交互命令及系统反馈的信息。在分析的基础上,进一步分析用户,确定交互的细节,包括指定窗口、设计窗口及设计报表形式等。人机交互部件在一定程度上依赖于所使用的图形用户接口,接口不同,人机交互部件的类型也不同。

具体的方法是:先对用户进行分类,可按照技能分为初级、中级和高级三类,也可按照组织级别分为总经理、部门经理和一般职员等,还可按照其他方式进行分类;然后,按照用户分类对用户的信息(特征、年龄、文化程度、技能水平、主要任务等)做进一步的描述;进而对人机交互的命令、命令层次进行设计。

人机交互部件的设计要遵循一致性原则,包括术语、步骤和动作的一致性,尽量减少用户操作的步骤,及时提供反馈信息,使人机交互界面易学,使用方便,富有吸引力。

(3) 任务管理部件 (Task Management Component)

任务管理是指确定各种类型的任务,并把任务分配到硬件或软件上去执行。为了划分任务,首先要分析并发性。通过 OOA 建立的动态模型是分析并发性的主要依据。通常把多个任务的并发执行称为多任务。

常见的任务有事件驱动型任务、时钟驱动型任务、优先任务、关键任务和协调任务等。

(4) 数据管理部件 (Data Management Component)

数据管理部件是系统存储、管理对象的基本设施,它建立在数据存储管理系统基础之上,并且独立于各种数据管理模式。设计数据管理部件,既需要设计数据格式,又需要设计相应的服务。设计数据格式的方法与所使用的数据存储管理模式密切相关,通常有文件系统和数据库管理系统两类数据存储模式。

4.5.3 对象模型技术 OMT

由 J.Umbaugh 和他的 4 位合作人于 1991 年推出的面向对象的方法学,又称为对象模型技术 (Object Model Technology, OMT)。作为一种软件工程方法学,它支持整个软件生存周期,覆盖了问题构成、分析、设计和实现等阶段。

OMT 方法体现了建模的思想,讨论如何建立一个实际的应用模型,从三个不同而又相关的角度建立三类模型:对象模型、动态模型和函数模型。OMT 为每一类模型提供了图形表示。

1. 对象模型技术的基本概念

OMT 方法讨论的核心问题就是建立对象模型、动态模型和函数模型三类模型。

对象模型描述了由对象和相应实体构成的系统静态结构,描述了系统中对象的标识、属性、操作及对象的相互关系。该模型使用对象图来描述,它是分析阶段三类模型的核心,提供了其他两类模型都适用的框架。

动态模型根据事件和状态描述了系统的控制结构,以及系统中与时间和操作顺序有关的内容,如标记变化的事件、事件的顺序、定义事件背景的状态等。

函数模型着重描述系统中与值的转换有关的问题,如函数、映射、约束和函数作用等。

三类模型描述的角度不同,却又相互联系。



2. 建立对象模型的 5 个层次

(1) 确定类与对象

类和对象是在问题域中客观存在的,系统分析员的主要任务就是通过分析找出这些类和对象。

(2) 确定关联

两个或多个对象之间的相互依赖,相互作用的关系就是关联,分析确定关联,要考虑问题域的边缘情况。

(3) 划分主题

将大型、复杂系统进一步划分成为不同的主题,以降低系统的复杂性。

(4) 确定属性

属性是对象的性质,一般确定属性的过程包括分析和选择两个步骤。

(5) 识别继承关系

确定了类中应该定义的属性之后,就可以利用继承机制共享公共性质,并对系统中众多的类加以组织。一般可使用自底向上和自顶向下两种方式建立继承关系。

3. 对象与类的描述

对象与类是构成对象模型的基本元素,图 4-23 给出了类的一般描述形式,由类名、属性和操作三部分组成。如图 4-24 所示,属性和操作还可做进一步的描述。图 4-25 所示则是对象的一般描述。

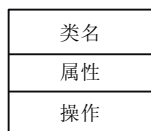


图 4-23 类的一般描述

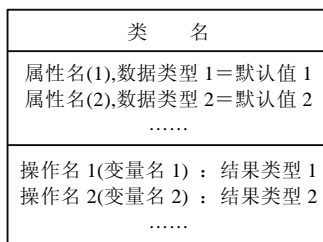


图 4-24 类的进一步描述



图 4-25 对象的一般描述

在对类或者对象的进一步描述中:

◎ 属性——属性名、补充细节;

◎ 操作——应用于类中对象或被类中对象使用的一种功能或者转换,包含操作允许的参数。

4. 链和关联

在对象模型中,类之间或对象之间的联系用关联(Association)或者链(Link)来描述。关联和链都是用一条直线连接相关的类或对象,直线上面标注关联和链的名称。

(1) 关联与链的关系

链——表示两个(或多个)对象之间的关系。

关联——描述具有公共结构和语义的一组链,表示类之间的关系。

关联描述两个或多个类之间的关系,链则是关联的实例。它们之间的关系与类和对象之

间的关系类似。在程序设计中，关联常用一个对象到另一对象的指针实现。

如图 4-26 所示，关联名是“国家有首都”，而联系对象“加拿大”与“渥太华”、“美国”“华盛顿”之间的链是“国家有首都”。

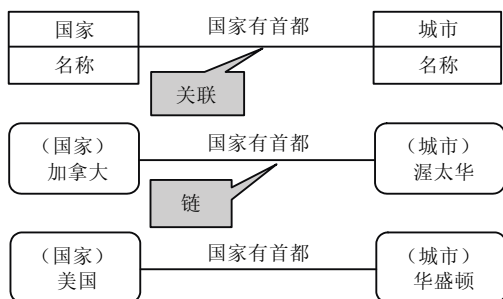


图 4-26 链与关联

如图 4-27 所示，可用二元关联、三元关联及多元关联来描述两个类、三个类及多个类之间的关系。

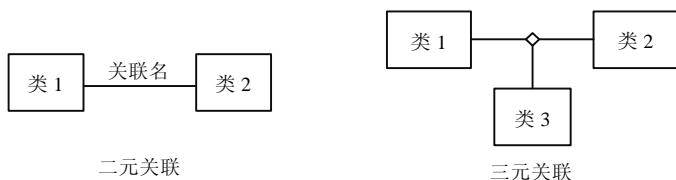


图 4-27 关联的表示

（2）关联的描述

关联还可用重数及链属性等做进一步的描述。

① 重数

重数也称关联的多重性，表示一个类中有多少个实例与一个相关类的某一个例子有关。重数限定了相关对象的个数。重数有以下表示方式：

● 表示“多个”，表示 0 或多个；

○ 表示“可选”，表示 0 或者 1。

也可在连线上标注数字表示重数：

1——只有 1 个；

1+——1 个或多个；

3~5——3~5 个之间。

图 4-28 表示一个公司有多个人。图 4-29 既表示一个公司有多个人，还表示一个人可属于多公司。

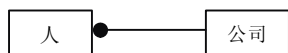


图 4-28 一对多的关联

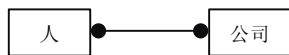


图 4-29 多对多的关联

② 链属性与角色

链属性是指关联中链的性质，链的每一个属性都有一个值。链属性的 OMT 符号是一个



方框，用一个弧形与两个类的直线和方框相连。方框内表示一个或多个属性。如图 4-30 所示，链属性为“访问许可”，还可以进一步用文字说明用户对文件的可访问性。

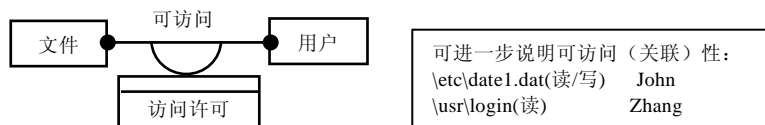


图 4-30 多对多关联的链属性

角色表示类在关联中的作用或职责，角色名用于唯一地标识端点。图 4-31 中给出了角色的描述，“公司”类在雇用关联中扮演的角色是雇主，而“个人”类的角色是雇员。

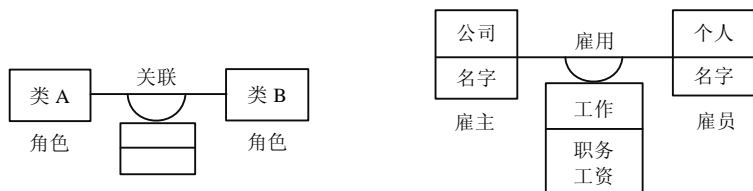


图 4-31 链属性与角色

5. 聚集和继承

(1) 聚集关系

聚集代表整体与部分的关系，聚集的符号是在整体的一端加上一个菱形框。图 4-32 中表示段落有多个句子。还可构成不同层次的多级聚集关系，图 4-33 表示一个微机系统的多级聚集。



图 4-32 聚集关系

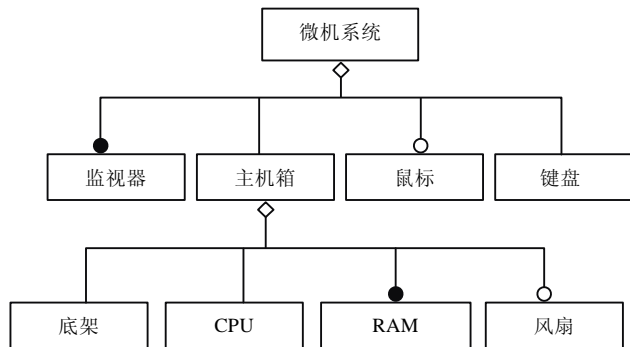


图 4-33 聚集的层次关系

(2) 继承关系

继承是指使用现存的定义作为基础，建立新定义的技术。继承性表示基类与子类的合作关系。子类继承了基类的特征。在 OMT 中，继承性用一个三角形表示。

继承分为单重继承和多重继承。单重继承只有一个父类，仅有单重继承的类层次结构是树层次结构。多重继承是指子类继承了多个父类的性质，是比单重继承更复杂的一般化关系，具有多重继承的层次结构是网状层次结构。图 4-34 中，“水陆两用船”继承了“陆上运输工具”和“水上运输工具”的属性与操作，它们之间是一种多重继承关系。

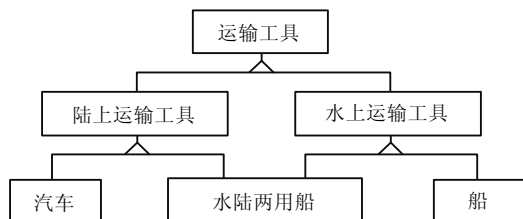


图 4-34 继承关系举例

6. 建立动态模型

动态模型着重于系统的逻辑结构，描述某时刻对象及其联系的变化。OMT 的动态图包括状态图和事件追踪图。

(1) 事件和状态

状态是指对象所具有的属性值，具有时间性和持续性。事件是指对象的触发行为，是从一个对象到另一个对象的信息的单向传递。脚本是指在系统的某一执行期间内的一系列事件。

在系统中具有属性值、链路的对象，可能相互激发，引起状态的一系列变化。有的事件传递的是简单信号，有的事件则传递的是数据值。由事件传送的数据值称为“属性”。

(2) 状态图

状态图是一个状态和事件的网络，侧重于描述每一类对象的动态行为，反映了由于事件引起的状态的迁移。如图 4-35 所示，状态用圆角矩形表示，框内标注状态名；事件用带箭头的连线表示；初始状态用圆点表示；终止状态用圆圈内加一个圆点表示。图 4-36 所示为一个打电话的状态图。

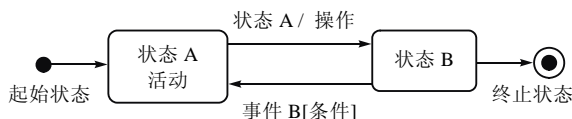


图 4-35 状态图

(3) 事件追踪图

事件追踪图侧重描述系统执行过程中一个特定的“场景(Scenarios)”。场景有时也叫“脚本”，是完成系统某个功能的一个事件序列。事件追踪图描述多个对象的集体行为。

脚本是系统某一次特定运行时期内发生的事件序列，例如，打电话的场景可以描述为：

- | | |
|------------|------------|
| ① 拿起电话受话器 | ② 电话忙音开始 |
| ③ 拨电话号码数 8 | ④ 电话忙音结束 |
| ⑤ 拨电话号码数 5 | ⑥ 拨电话号码数 5 |
| ⑦ 拨电话号码数 1 | ⑧ 拨电话号码数 2 |
| ⑨ 拨电话号码数 5 | ⑩ 拨电话号码数 6 |

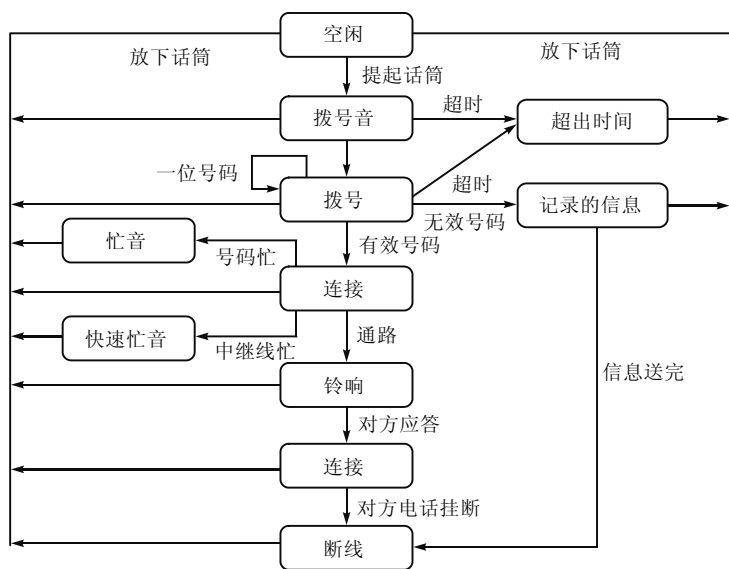


图 4-36 打电话的状态图

- | | |
|-------------|--------------|
| ⑪ 拨电话号码数 7 | ⑫ 接电话方电话开始振铃 |
| ⑬ 打电话方听见振铃声 | ⑭ 接电话方接电话 |
| ⑮ 接电话方停止振铃 | ⑯ 打电话方停止振铃声 |
| ⑰ 通电话 | ⑱ 接电话方挂电话 |
| ⑲ 电话切断 | ⑳ 打电话方挂电话 |

按照对打电话的场景的描述，图 4-37 画出了打电话的事件追踪图。

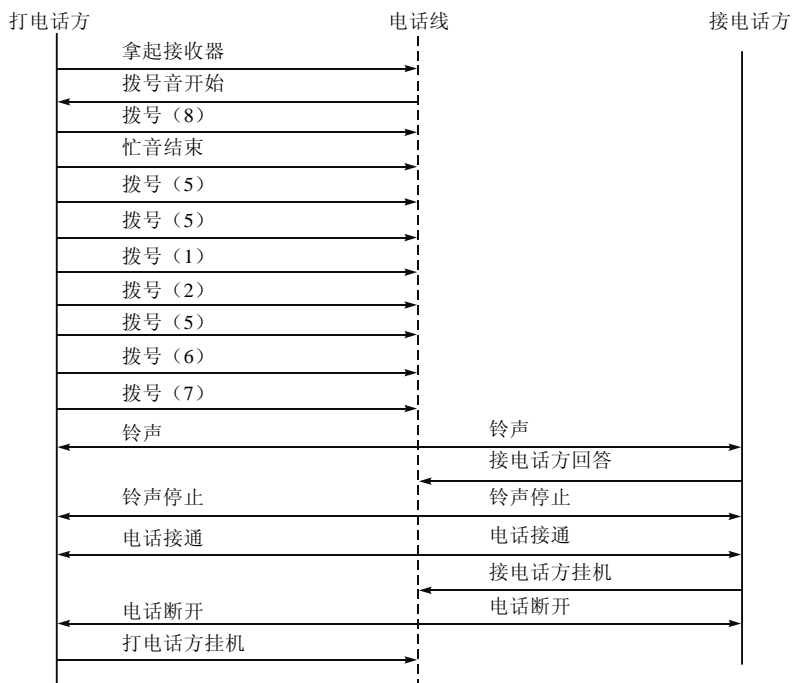


图 4-37 打电话的事件追踪图

4.5.4 OOSE方法

面向对象软件工程（Object-Oriented Software Engineering, OOSE）方法是 1992 年由 Jacobson 在其出版的专著《面向对象的软件工程》中提出的。OOSE 方法采用 5 类模型来建立目标系统。

① 需求模型（Requirements Model, RM）。用于获取用户的需求，识别对象，主要的描述手段有用例图（Use Case）、问题域对象模型及用户界面。

② 分析模型（Analysis Model, AM）。该模型定义系统的基本结构，将 RM 中的对象，分别识别到 AM 的实体对象、界面对象和控制对象三类对象中。每类对象都有自己的任务、目标并模拟系统的某个方面。

实体对象模拟那些在系统中需要长期保存并加以处理的信息，实体对象由使用事件确定，通常与现实生活中的一些概念符合。界面对象的任务是提供用户与系统之间的双向通信，在使用事件中所指定的所有功能都直接依赖于系统环境，它们都放在界面对象中。而控制对象的典型作用是将另外一些对象组合形成一个事件。

③ 设计模型（Design Model, DM）。AM 只注重系统的逻辑构造，而 DM 需要考虑具体的运行环境，将分析模型中的对象定义为模块。

④ 实现模型（Implementation Model, IM）。即使用面向对象的语言来实现系统。

⑤ 测试模型（Testing Model, TM）。测试的重要依据是 RM 和 AM，测试的方法与技术与第 8 章软件测试中所介绍的类似，而底层是对类（对象）的测试。TM 实际上是一个测试报告。

如图 4-38 所示，OOSE 的开发活动主要分为三类：分析、构造和测试。其中，分析过程分为需求分析（Requirements Analysis）和健壮分析（Robustness Analysis）两个子过程。分析活动分别产生需求模型和分析模型。构造活动包括设计（Design）和实现（Implementation）两个子过程，分别产生设计模型和实现模型。测试过程包括单元测试（Unit Testing）、集成测试（Integration Testing）和系统测试（System Testing）三个过程，它们共同产生测试模型。

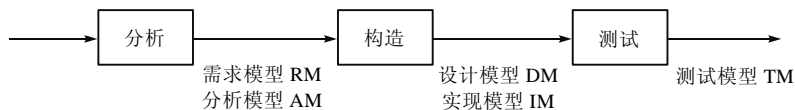


图 4-38 OOSE 的开发活动

OOSE 对面向对象方法的发展和进步有两大贡献：一是提出了用例（Use Case）这个重要概念，用例定义需求，提供了很好的需求分析策略和描述手段，弥补了以前的面向对象需求中的缺陷；另一个重大贡献是定义了交互图，交互图对一组相互协作的对象在完成一个用例时执行的操作及它们之间传递的消息和时间顺序做了更精确的描述。

在开发各种模型时，用例是贯穿 OOSE 活动的核心，描述了系统的需求及功能。用例实际上描述的是系统用户（也称使用者）对于系统的使用情况，是从使用者的角度来确定系统的功能。

因此，首先必须分析确定系统的使用者，然后进一步考虑使用者的主要任务、使用的方



式，识别所使用的事件，即用例。如图 4-39 所示，用人形表示使用者，用椭圆表示用例，用大矩形框表示系统的边界，用连接使用者和用例的箭头线表示使用者驱动事件的完成。

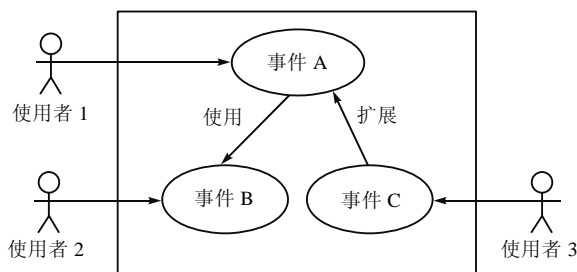


图 4-39 用例图

习题 4

- 4.1 与传统程序设计模式中的过程调用相比，消息传递机制有何本质区别？
- 4.2 比较面向对象方法与结构化方法的特点，说明为什么面向对象方法比结构化方法更加优越。
- 4.3 OOA 方法分析过程和构造 OOA 概念模型的顺序由 5 个层次组成，请简述这 5 个层次。
- 4.4 面向对象的分析包括哪些主要活动？所建立的分析模型包括哪些类型的模型？
- 4.5 面向对象设计的主要任务是什么？
- 4.6 为什么面向对象的方法能够有效地解决软件需求中存在的问题？
- 4.7 OMT 方法明确提出了建模的概念，说明为什么在软件开发过程中需要进行建模？
- 4.8 当重要的对象被发现后，通过一组互相关联的模型详细表示类之间的关系和对象的行为，这些模型从 4 个不同的侧面表示了软件的体系结构、静态逻辑、动态逻辑、静态物理和动态物理。试描述一下这 4 种特性。
- 4.9 为什么说面向对象的方法为软件复用提供了良好的环境？

第 5 章 统一建模语言（UML）

UML 是软件界第一个统一的可视化的建模语言，已成为国际软件界广泛承认的标准，应用领域很广泛，可用于商业建模（Business Modeling）、软件开发建模的各个阶段，也可用于其他类型的系统。它是一种通用建模语言，具有创建系统的静态结构和动态行为等多种结构模型的能力，具有可扩展性和通用性，适合于多种、多变结构的建模。



5.1 UML概述

软件工程领域在 1995 年至 1997 年取得了前所未有的进展，其成果超过软件工程领域 1995 年前 15 年的成就总和。其中最重要的、具有划时代重大意义的成果之一就是统一建模语言（Unified Modeling Language, UML）。

UML 的价值在于它综合并体现了世界上面向对象方法实践的最好经验，支持用例驱动（Use-Case Driven），以架构为中心（Architecture-Centric），递增（Incremental）和迭代（Iterative）地进行软件开发。因此，在世界范围内，至少在今后 10 年内，UML 将是面向对象技术领域内占主导地位的标准建模语言。



5.1.1 UML的形成

从 20 世纪 80 年代初期开始，众多的方法学家都在尝试用不同的方法进行软件分析和设计，80 年代末，形成以 Smalltalk 语言为代表的第一代面向对象的方法。到了 90 年代中期，出现了第二代面向对象方法，具有代表性的有：G.Booch 的面向对象的开发方法，P.Coad 和 E.Yourdon 的面向对象的分析（OOA）和面向对象的设计（OOD），J.Rumbaugh 等人的对象建模技术（OMT）及 Jacobson 的面向对象的软件工程（OOSE）等。此时，面向对象的方法已经成为软件分析和设计的主流方法。

1994 年 10 月，J.Rumbaugh 和 G.Booch 共同合作，把 OMT 和 Booch 方法统一起来，于 1995 年推出一个称为“统一方法”（Unified Method, UM）的 0.8 版。随后，Jacobson 加入，并采用他的用例（User Case）思想，在 1996 年，推出“统一建模语言”0.9 版。

1997 年 1 月，UML 1.0 版被正式提交给国际对象管理组织（Object Management Group, OMG），作为软件建模语言标准的候选。在其后的半年多时间里，一些重要的软件开发商和系统集成商，如 IBM、Microsoft、HP 等 700 多家公司，都成为“UML 伙伴”。

1997 年 11 月 7 日，UML 1.1 版被 OMG 正式批准为基于面向对象的技术的标准建模语言。UML 的形成过程如图 5-1 所示。

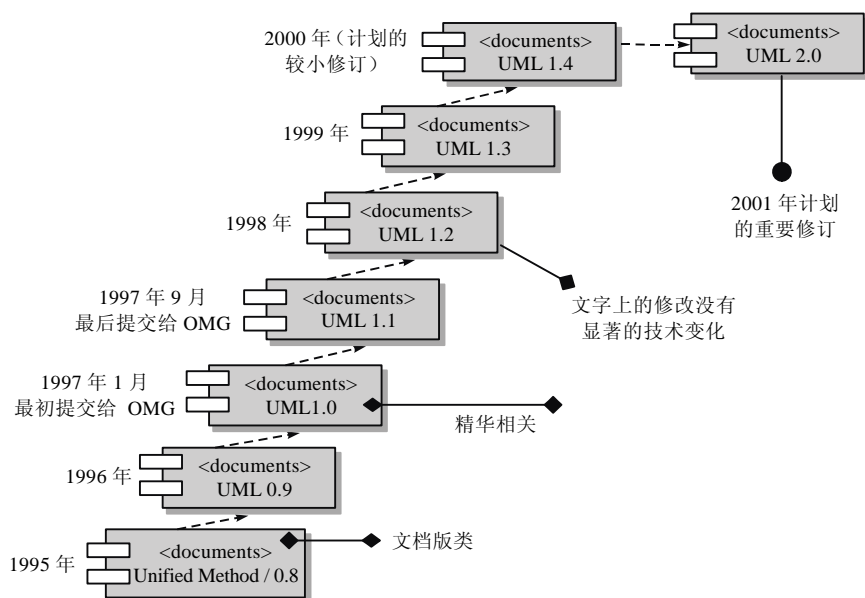


图 5-1 UML 的形成过程



5.1.2 UML的特点

(1) 统一标准

UML 统一了面向对象的主要流派 Booch、OMT 和 OOSE 等方法中的基本概念，已成为 OMG 的正式标准，并提供了标准的面向对象的模型元素的定义和表示。统一的标准，有利于面向对象方法的应用和发展。

(2) 面向对象

UML 还吸取了面向对象技术领域其他流派的长处。UML 符号表示考虑了各种方法的图形表示，删掉了大量易引起混乱的、多余的和极少使用的符号，也添加了一些新符号，可以说是集面向对象技术的众家之长。

(3) 可视化、表示能力强

系统的逻辑模型或实现模型都能用 UML 的可视化模型清晰地表示出来，对系统描述能力强，模型蕴涵的信息丰富，可用于复杂软件系统的建模。

(4) 独立于过程

UML 是系统建模语言，独立于开发过程。

(5) 易掌握、易用

UML 的概念明确，建模表示法简洁明了，图形结构清晰，易于掌握使用。



5.1.3 UML建模及其构成

模型是一个系统的完整的抽象，是人们对某个领域特定问题的求解及解决方案，对它们的理解和认识都蕴涵在模型中。在各种工程问题中，建模已成为工程实践的重要组成部分。

例如，在建筑行业，要修建一幢大楼前，先要按照一定比例建立该大楼的模型，使建筑师和用户对未来大楼的外形、特性等有一个感性认识，便于进一步修改、完善、评价或审批该建筑方案。

1. 软件开发过程中建模的必要性

鉴于软件系统的复杂性和规模的不断增大，需要建立不同的模型对系统的各个层次进行描述。软件模型一般包括数学模型、描述模型和图形模型。

由于 UML 以图形模型为主，模型的直观性及丰富的信息描述，便于开发人员与用户的交流。

模型为以后的系统维护和升级提供了文档。

UML 是一种标准的图形化（可视化）的建模型语言，UML 的核心是建立系统的各类模型。通常，开发一个计算机软件系统是为了解决某个领域的特定问题，问题的求解过程，就是从领域问题到计算机系统的映射过程。图 5-2 描述了软件项目开发的建模过程。

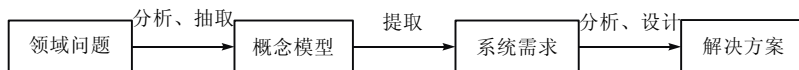


图 5-2 软件项目开发的建模过程

2. UML的主要内容

UML 的主要内容包括 UML 的语义和 UML 的表示法。

(1) UML 语义

UML 的语义通过元模型来精确定义。元模型为 UML 的所有元素在语法和语义上提供了简单、一致、通用的定义性说明，使开发者能在语义上取得一致，消除了因人而异的表达方法所造成的影响。此外，UML 还支持对元模型的扩展定义。

UML 支持各种类型的语义，如布尔、表达式、列表、阶、名字、坐标、字符串和时间等，还允许用户自定义类型。

(2) UML 表示法

UML 表示法定义了图形符号的表示，为开发者或开发工具使用这些图形符号和文本语法进行系统建模提供了标准。这些图形符号和文字所表达的是应用级的模型，在语义上它是 UML 元模型的实例。

UML 表示法由通用表示和图形表示两部分组成。

① 通用表示

字符串：表示有关模型的信息。

名字：表示模型元素。

标号：赋予图形符号的字符串。

特定字符串：赋予图形符号的特性。

类型表达式：声明属性变量及参数。

定制：是一种用已有的模型元素来定义新模型元素的机制。



② 图形表示

UML 的图形表示将在 5.1.4 节详细介绍。

5.1.4 UML的图形表示

UML 建模型语言的描述方式以标准的图形表示为主,是由视图(Views)、图(Diagrams)、模型元素(Model Elements)和通用机制(General Mechanism)构成的层次关系。

1. 视图(Views)

一个系统应从不同的角度进行描述,视图就是从不同的视角观察和建立的系统模型图。一个视图由多个图构成,图不是一个图表(Graph),而是在某一个抽象层上对系统的抽象表示。

描述一个系统需要定义一定数量的视图,每个视图表示系统的一个特殊方面或者系统的某个特性。多个视图才能建立一个完整的系统模型图。另外,视图还把建模语言和系统开发时选择的方法或过程连接起来。图 5-3 描述了常用的 UML 视图。

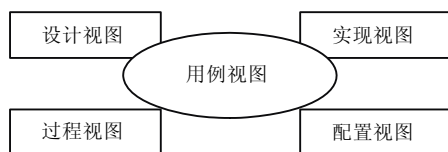


图 5-3 常用的 UML 视图

(1) 用例视图(Use Case View)

从使用者的角度描述系统的外部特性及系统应具备的功能等。用例视图是其他视图的核心和基础,直接影响到其他视图的建立和描述。

(2) 设计视图(Design View)

设计视图用于描述系统设计特征和系统内部的结构,包括结构模型视图和行为模型视图,前者描述系统的静态结构,包括类图、对象图,后者描述系统的动态行为,包括交互图、状态图和活动图。

(3) 过程视图(Process View)

过程视图表示系统内部的控制机制和并发特征。常用类图描述过程结构,用交互图描述过程行为。

(4) 实现视图(Implementation View)

表示系统的实现特征,常用构件图表示,由一些独立的构件(文件)构成,描述了系统的软件特性。

(5) 配置视图(Deployment View)

配置视图用于描述系统的物理配置特征,系统的物理架构。另外,根据系统的应用领域和特性,还可以建立其他视图。

2. 图（Diagrams）

图用来描述一个视图的内容，是构成视图的成分。UML 语言定义了 9 种不同的图，包括用例图、类图、对象图、包图、状态图、活动图、顺序图、合作图、构件图及部件图，将它们有机地结合起来就可以描述系统的所有视图。在 UML 建模语言中又把这 9 种图分为 5 类。

（1）用例图（Use Case Diagram）

用例图从用户角度描述系统应该具备的功能，并指出各功能的操作者。

（2）静态图（Static Diagram）

静态图表示系统的静态结构，包括类图、对象图和包图。

（3）行为图（Behavior Diagram）

行为图描述系统的动态模型和组成对象间的交互关系，包括状态图和活动图。

（4）交互图（Interactive Diagram）

交互图描述对象间的交互关系，包括顺序图和合作图。

（5）实现图（Implementation Diagram）

实现图用于描述系统的物理实现和物理配置，包括构件图及部件图。

3. 模型元素（Model Elements）

模型元素代表面向对象中的类、对象、关系和消息等概念，是构成图的最基本的元素。无论在各类图中使用它，模型元素总是具有相同的含义和相同的符号表示。模型元素是 UML 建模的最基本的成分，将在 4.1.5 节中进行详细介绍。

4. 通用机制（General Mechanism）

通用机制用于表示其他信息，如注释、模型元素的语义等。另外，为了适应用户的需求，它还提供了扩展机制（Extensibility Mechanism），包括构造型（Stereotype）、标记值（Tagged Value）和约束（Constraint）等，使用 UML 语言能够适应一种特殊的方法（或过程），或扩充至一个组织或用户。

例如，注释用于对 UML 语言的元素或实体进行说明、解释和描述。通常，用自然语言进行注释。在 UML 的各种模型图中，凡是需要注释的元素或实体均可加注释。

注释由注释体和注释连接组成。注释体的图符是一个右上角翻下的矩形，其中标注要注释的内容。注释连接用虚线表示，它把注释体与被注释的元素连接起来。如图 5-4 所示，“这是一个类”为注释体，对类“人员”进行注释。

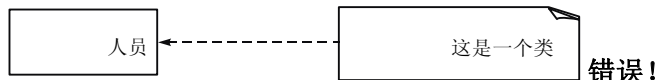


图 5-4 注释

5.1.5 通用模型元素

可以在图中使用的概念、对象等统称为模型元素。一个模型元素可以在多个不同的图中



使用，所以又称为通用模型元素。模型元素是 UML 构建各种模型的基本单位。通用模型元素分为以下两类。

① 基元素是指已由 UML 定义的模型元素，如：类、结点、构件、注释、关联、依赖和泛化等。

② 构造型元素是在基元素的基础上所构造的新的模型元素，通常是由基元素增加了新的定义而构成的，如扩展基元素的语义（不能扩展语法结构），也允许用户自定义。构造型元素用括在双尖括号<<>>中的字符串表示。

目前，UML 提供了 40 多个预定义的构造型元素，如<<include>>、<<extend>>等。

模型元素在图中用其相应的图形符号表示。常用的模型元素符号如图 5-5 所示，图中给出了类、对象、结点、包和组件等部分常用的模型元素的符号图例。利用模型元素可以把图形直观地表示出来，一个元素（符号）遵循一定的规则，可存在于多个不同类型的图中。

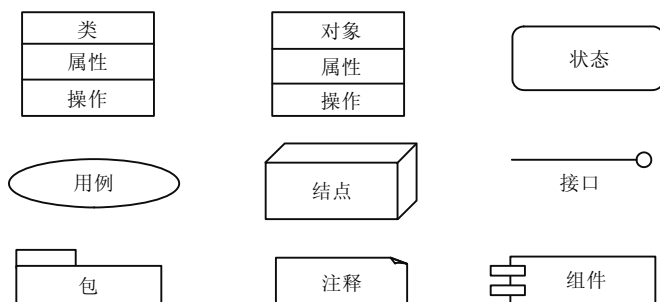


图 5-5 模型元素

特别要说明的是，模型元素与模型元素之间的连接关系也是模型元素。常见的连接关系有关联（association）、泛化（generalization）、依赖（dependency）和聚合（aggregation）等，其中，聚合是关联的一种特殊形式。这些连接关系的图形符号如图 5-6 所示。

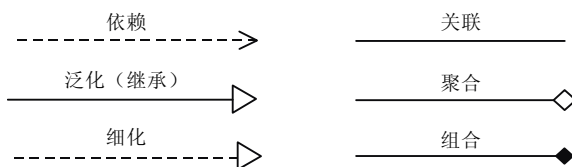


图 5-6 连接关系的图形符号

（1）关联

关联是一种最常见的连接关系，用于连接（connect）模型元素或链接（link）实例。

（2）依赖

依赖表示一个元素以某种方式依赖于另一个元素。依赖关系描述的是两个模型元素（类、组合、用例等）之间的语义上的连接关系，其中一个模型元素是独立的，另一个模型元素是非独立的（或依赖的），它依赖于独立的模型元素。如图 5-7 所示，类 A 依赖于类 B，其依赖关系为<<友元>>（friend），其中，友元是依赖关系变体（varieties）的一种，是预定义的构造型元素，允许一个元素访问另一个元素，不管被访问的元素是否具有可见性。

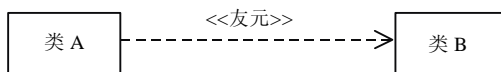


图 5-7 依赖关系

(3) 泛化

泛化表示一般与特殊的关系，即“一般”元素是“特殊”关系的泛化，常用于描述父类与子类之间的继承关系，如图 5-8 所示。



图 5-8 泛化关系

(4) 聚合

聚合表示整体与部分的关系，即由部分元素构成整体。如图 5-9 所示，类 A 是整体类，类 B 是部分类。

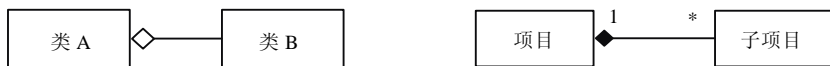


图 5-9 聚合关系

另外，组合也是一种聚合关系，它的“整体”与“部分”之间的关系比一般聚合更加紧密。“*”表示有多个子项目。

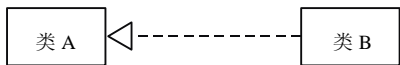


图 5-10 细化

细化是依赖关系的一个变体，描述两个不同语义层次上的元素之间的关系。细化与类的抽象层次有密切的关系，人们在构造模型时不可能一下子就把模型完整、准确地构造出来，而是要经过逐步细化的过程，也就是逐步求精的过程。图 5-10 表示类 B 是类 A 细化的结果。



5.2 建立用例模型

1992 年，由 Jacobson 提出了 Use Case 的概念及可视化的表示方法——Use Case 图，受到了 IT 界的欢迎，被广泛应用于面向对象的系统分析中。用例驱动的系统分析与设计方法已成为面向对象的系统分析与设计方法的主流。

用例建模技术，从用户的角度描述系统的功能需求，在宏观上给出模型的总体轮廓，通过对典型用例的分析，使开发者能够与用户进行充分交流，有效地了解和获取用户需求，分析确定系统的需求。

UML 的用例模型一直被推荐为识别和捕获需求的首选工具。同时它驱动了需求分析之后各阶段的开发工作，不仅在开发过程中保证了系统所有功能的实现，而且被用于验证和检测所开发的系统，从而影响到开发工作的各个阶段和 UML 的各类模型。



5.2.1 需求分析与用例建模

用例模型（Use Case Model）描述的是外部执行者（Actor），如用户，所理解的系统功能。它描述的是一个系统做什么（What），而不是说明怎么做（How）。用例模型不关心系统设计。

虽然用例模型主要用于需求分析阶段，但它的建立是系统开发者和用户反复讨论的结果，表明了开发者和用户对需求规格达成的共识。

用例模型由若干个用例图构成，在 UML 中，构成用例图的主要元素是用例和执行者及其之间的联系。

图 5-11 描述了“医院病房监护系统”的高层用例图。该用例图是在 2.5.1 节对系统进行需求分析的基础上，从值班护士、医生、病人等用户角度提出系统的主要功能而建立的，确定了“中央监护”、“病症监护”等用例。用例所执行的功能则是由相应的执行者来驱动的。图中的矩形框表示系统的范围，不是用例图的必要成分。

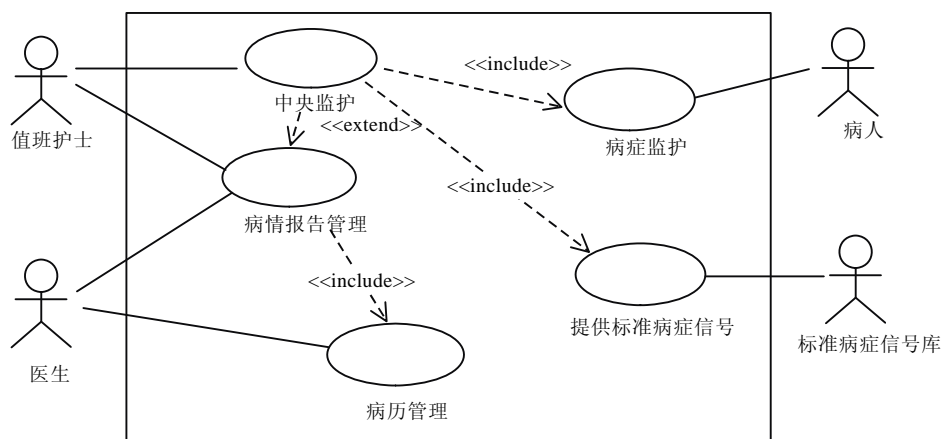


图 5-11 “医院病房监护系统”高层用例图

建立系统用例模型的过程就是对系统进行功能需求分析的过程。图 5-12 描述了用例建模的过程。



图 5-12 用例建模过程

- ① 定义系统，确定系统范围，获取、分析系统功能需求。
 - ② 确定执行者和用例：执行者通常是指使用系统功能的外部用户或系统，用例则是一个子系统或系统的一个独立、完整功能。
 - ③ 描述执行者和用例关系：确定各模型元素之间的关联、包含、扩展及泛化等关系。
 - ④ 确认模型：确认用例模型与用户需求的一致性，通常由用户与开发者共同完成。
- 显然，确定执行者和用例是建立用例模型的关键，下面将做进一步介绍。

5.2.2 确定执行者

用例图中包含执行者、用例和连接等三种模型元素，如图 5-13 所示。执行者（Actor）是指用户在系统中所扮演的角色，执行者也称为角色。执行者在用例图中是用类似人的图形来表示的，但执行者未必是人。例如，执行者也可以是一个外界系统，该外界系统可能需要从当前系统中获取信息，与当前系统进行交互。

图 5-14 描述了自动售货机系统的用例图，要售货，执行者必须有“顾客”和“供货人”，自动售货机虽是自动收款，但每天要有专门的“收银员”负责取出所收货款。矩形框表示系统的边界，用于划分系统的功能范围。

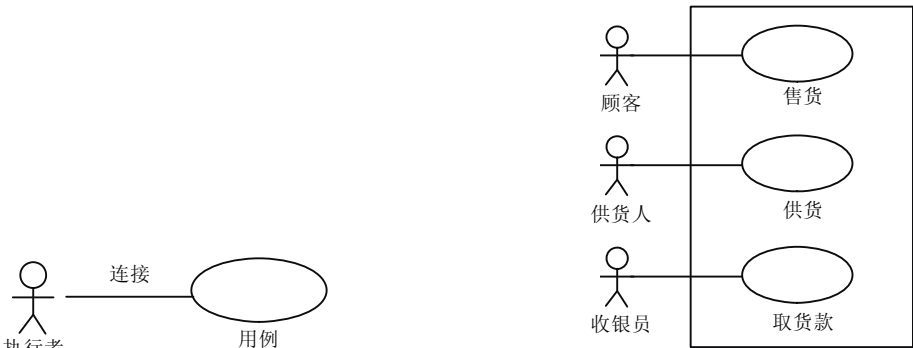


图 5-13 用例图的三种模型元素

图 5-14 自动售货机系统用例图

如何确定执行者是画用例图首要问题，首先要与系统的用户进行广泛而深入的交流，明确系统的主要功能，以及使用系统的用户责任等。此外，还可以通过回答以下问题来确定执行者。

- ① 谁使用系统的主要功能（主执行者）？
- ② 谁需要从系统获得对日常工作的支持和服务？
- ③ 需要谁维护管理系统的日常运行（副执行者）？
- ④ 系统需要控制哪些硬件设备？
- ⑤ 系统需要与其他哪些系统交互？
- ⑥ 谁需要使用系统产生的结果（值）？

识别出的角色应该用文字的形式或角色描述模板来做进一步的描述。角色描述模板如图 5-15 所示。

【例 5-1】有一个自动取款机（Auto Trade Machine，ATM）系统，为储户提供 24 小时的服务，储户需要提款时，必须将银行信用卡插入 ATM 中，并输入正确的口令后才能取款。通过回答下面的问题来识别角色。

- ① 谁使用 ATM 系统的主要功能（提款）？
答：储户。
- ② 谁需要从 ATM 系统获得对日常工作的支持和服务？
答：出纳员？（不肯定）

角色：_____

角色职责：

角色职责识别：

图 5-15 角色描述模板



③ 需要谁维护管理系统的日常运行?

答: 银行工作人员、系统工程师。

④ ATM 系统需要控制哪些硬件设备?

答: 银行信用卡。

⑤ 系统需要与其他哪些系统交互?

答: 不清楚。

⑥ 谁需要使用 ATM 系统产生的结果?

答: 银行会计、储户。

通过回答以上问题, 得到可能的角色有: 储户、出纳员、银行工作人员、系统工程师、银行信用卡、银行会计。

在此基础上对初始的角色进行分析和筛选。对于问题①、④、⑤、⑥的回答没有什么问题。

问题②的答案“出纳员”不能肯定, 应该做进一步分析, 经过分析, “出纳员”与 ATM 系统的关系不大, 他所完成的工作主要是手工操作, 而并非需要从 ATM 系统获得对日常工作的支持和服务, 因此可以除去。

问题③所确定的“银行工作人员”主要负责 ATM 信息的维护及 ATM 一般故障的维修, 而“系统工程师”则是掌握和熟悉 ATM 技术及系统配置的工程技术人员, 事实上对系统的日常维护工作, 只需要银行工作人员就行了。

图 5-16 储户角色描述模板

角色: 储户
角色职责:
插入信用卡
输入口令
输入交易金额
角色职责识别:
(1) 使用系统主要功能
(2) 使用系统运行结果

根据以上分析, 最后确定与 ATM 系统直接相关的执行者为:

储户、银行工作人员、银行信用卡、银行会计。图 5-16 给出了角色“储户”的描述模板。

5.2.3 确定用例

本质上讲, 一个用例是用户与计算机之间的一次典型交互作用。在 UML 中, 用例被定义成系统执行的一系列动作(功能), 即用例是对系统用户需求的描述, 表达了系统的功能和所提供的服务。UML 中的用例用椭圆表示, 用例的名字写在椭圆的下方或内部, 用例位于系统边界的内部, 角色与用例之间的关联关系(或通信关联关系)用一条直线表示。

1. 识别用例

识别用例的方法有多种, 其基本的出发点都是从系统的功能考虑, 例如, 可以通过回答以下问题来确定用例。

① 与系统实现有关的主要问题是什么?

② 系统需要哪些输入/输出? 这些输入/输出从何而来? 到哪里去?

③ 执行者需要系统提供哪些功能?

④ 执行者是否需要系统对系统中的信息进行读、创建、修改、删除或存储?

如果首先确定了系统的角色, 也可以通过角色来识别用例。

2. 用例的特征

根据用例特征来分析确定用例, 也是一种常用的确定用例的方法。用例的主要特征如下。

- ① 用例捕获某些用户可见的需求, 实现一个具体的用户目标。
- ② 用例由执行者激活, 即用例总是由执行者启动, 并提供确切的值给执行者。
- ③ 用例可大可小, 但它必须是对一个具体的用户目标实现的完整描述。

图 5-14 中确定的用例有: “售货”、“供货”和“取货款”。用例往往还需要进一步细化。

【例 5-2】分析 2.5.1 节所给出的实例——医院病房监护系统, 已确定该系统有以下角色: 值班护士、医生、病人和标准病症信号库, 识别用例, 并对用例做进一步分解和描述。

通过回答下面问题来识别用例。

- ① 与系统实现有关的主要问题是什么?

答: 中央监护, 将病人的病症信号与标准病症信号库里病症信号的正常值进行比较, 当病症出现异常时系统自动报警, 自动更新病历, 并打印病情报告。

- ② 系统需要哪些输入/输出? 这些输入/输出从何而来? 到哪里去?

答: 输入:

- ◎ 病症信号, 由用例“病症监护”将从病人采集到的病症信号, 实时地传送到中央监护系统中。
- ◎ 病症信号的正常值, 用例“中央监护”将病人的病症信号值与标准病症信号库中病症的正常值进行比较。

输出:

- ◎ 报警信号, 当病症出现异常时“中央监护”系统自动报警。
 - ◎ 病情报告, 根据医生要求或病症信号异常时, “病情报告管理”自动打印病情报告。
- ③ 执行者需要系统提供哪些功能?

答: 医生和值班护士需要查看病情报告、病历并进行打印。

- ④ 执行者是否需要系统对系统中的信息进行读、创建、修改、删除或存储?

答: 病人通过用例“病症监护”采集如血压、脉搏、体温等病症信号。

通过回答以上问题, 得到系统用例: 中央监护、病症监护、提供标准病症信号、病历管理、病情报告管理。

用例确定后, 还应该用文字或者用例描述模板进行描述。下面是对医院病房监护系统主要用例的分解和文字描述。

- ① 中央监护

- ◎ 分解信号。将从病症监护器传送来的组合病症信号分解为系统可以处理的信号。
- ◎ 比较信号。将病人的病症信号与标准信号进行比较。
- ◎ 报警。如果病症信号发生异常 (即高于峰值), 则发出报警信号 (警报声、灯光、大屏幕显示)。
- ◎ 数据格式化。将处理后的数据格式化以便写入病历库中。

- ② 病症监护

- ◎ 信号采集。采集病人的病症信号。
- ◎ 模数转化。将采集来的模拟信号转化为数字信号。
- ◎ 信号数据组合。将采集到的脉搏、血压等信号数据组合为一组信号数据。
- ◎ 采样频率改变。根据病人的情况改变监视器采样频率。



③ 提供标准病症信号（此用例不分解）

④ 病历管理

◎ 生成病历。将各种症信号经过格式化后，加上时间戳，存入病历库中。

◎ 查看病历。医生根据需要随时在计算机屏幕上查看病历。

◎ 更新病历。定时清理病历库，将陈旧的病历转储存档，以更新病历。

◎ 打印病历。定时打印病历，作为医生诊病的依据。

⑤ 病情报告

◎ 显示病情报告。在显示器上显示各种类型的病情报告。

◎ 打印病情报告。根据医生要求，在打印机上打印病情报告。在产生报警时，立即自动打印预定义的病情报告。

对用例可用类似的模板来进行描述，也可用文字进行描述。

与角色模板类似，确定用例后也应该用“用例模板”进行描述。用例模板在 UML 中并未给出规范，图 5-17 给出了用例“中央监护”的一种描述模板，对该用例的主要信息进行了说明。

用例名：中央监护	执行者：值班护士、医生
目标：对病人的病症信号进行监测、处理，超过极限报警	
功能描述： 1. 分解信号 将从病症监护器传送来的组合病症信号分解为系统可以处理的信号。 2. 比较信号 将病人的病症信号与标准信号进行比较。 3. 报警 如果病症信号发生异常（即高于峰值），则发出报警信号。 4. 数据格式化 将处理后的数据格式化以便写入病历库中。	
其他非功能需求：高可靠性、实时性	
主要步骤： 1. 按设定频率连续接收来自各病人的病症信号，并进行分解。 2. 将病人的病症信号与专家系统（标准病症信号库）中的标准信号进行比较，判断是否超过极限值。 3. 若超过极限值，则报警，并及时更新病历和打印病情报告。	
相关用例：病症监护、提供标准病症信号、病历管理、病情报告管理。	
相关信息： 优先级：报警处理具有最高优先级 3，一般病历管理为 1，其他为 2。 性能：实时性、高可靠性。 执行频率：根据病情严重程度 12~30 次/小时。	

图 5-17 用例“中央监护”一种描述模板



5.2.4 建立用例之间的关系

用例除了与执行者有联系外，用例之间也存在一定的联系，用例之间通常有关联、包含、扩展及泛化等关系，包含和扩展是构造型元素，分别用<<include>>和<<extend>>表示，图形表示为一个虚线箭头。在新版的 UML 中，“包含”<<include>>替代了“使用”<<uses>>。

1. <<include>>关系

<<include>>本质上是一种使用关系，当一个用例包含另一个用例时，这两个用例之间就构成了使用关系。图 5-18 所示的自动售货机系统中，“供货”与“取货款”这两个用例的

开始动作都是打开机器，而它们最后的动作都是关闭机器。因此，将开始动作抽象为“打开机器”用例，将最后的动作抽象为“关闭机器”用例，“供货”与“取货款”用例在执行时必须包含这两个用例。

2. <<extend>>关系

扩展<<extend>>则是向一个用例中加入一些新的动作后构成了另外一个用例，这两个用例之间的关系就是扩展关系。后者是继承前者的一些行为得来的，通常把前者称为基本用例，而把后者称为扩展用例。基本用例通常是一个独立的用例，一个扩展用例是对基本用例在对某些“扩展点”（Extension Points）功能的增加。被扩展的用例是一般用例，扩展的用例是特殊用例。例如，在图 5-18 中，“售货”是一个基本用例，定义的是出售罐装饮料，而用例“售散装饮料”则是在继承了“售货”的一般功能的基础上进行修改，因此是“售货”的扩展。

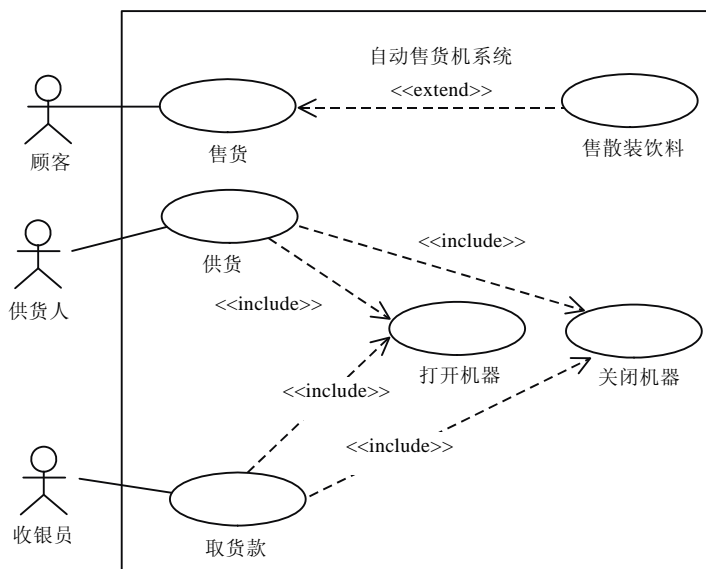


图 5-18 含有包含和扩展关系的用例图

用例模型是获取需求、规划和控制项目迭代过程的基本工具。在建立用例模型时还要考虑用例的数目，若用例数目多，则每个用例较小，小的用例易执行，但若用例数量过多，则会使用例图显得过于繁杂。因此要根据系统大小，适当选择用例数目。

5.2.5 用例建模实例

【例 5-3】建立金融贸易系统的用例图。

按照获取执行者的方法，确定该系统的 4 种执行者：贸易经理、营销人员、销售人员和记账系统。贸易经理的职责是确定“边界”，所谓边界，是指金融贸易的范围。

在该系统所确定的用例中，基本的用例是“进行交易”，之所以将“评价”作为一个独立的用例，是因为用例“交易估价”与“风险分析”都要使用公共的“评价”动作，都要对交易进行评价，因此，“评价”用例被“交易估价”与“风险分析”所包含。

在交易过程中，可能会出现超越交易范围或超过交易量的情况，这时，允许对用例“进



行交易”进行修改、扩充，即用例“超越边界”是用例“进行交易”的扩展。

虽然大多数执行者是人，但是，图 5-19 中执行者“记账系统”是一个外部系统，它需要与用例“更新账目”进行交互。

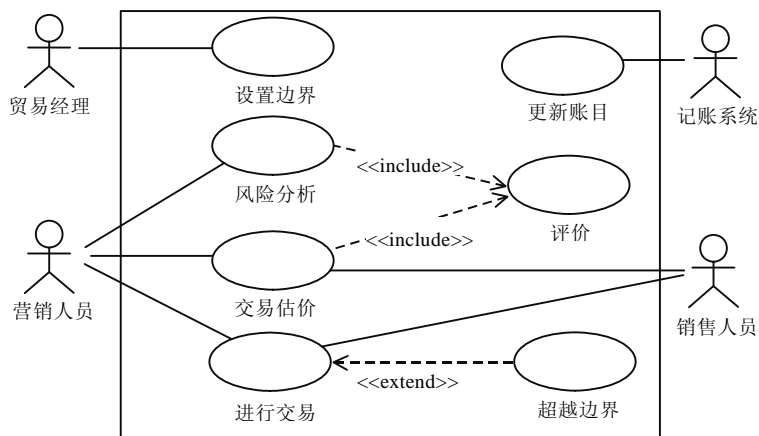


图 5-19 金融贸易系统用例图

【例 5-4】建立项目与资源管理系统（PRMS）的用例图。系统的主要功能是：项目管理、资源管理和系统管理。项目管理包括项目的增加、删除、更新；资源管理包括对资源和技能的添加、删除和更新；系统管理包括系统的启动和关闭，以及数据的存储和备份等功能。

1. 分析确定系统的执行者（角色）

执行者是对系统外的对象描述，是用户作用于系统的一个角色，它有自己的目标，通过与系统的交互来实现，交互包括信息交换及与系统的协同。

执行者可以是人，也可以是一个外部系统。通过回答 5.2.2 节中的问题，可以确定本系统的 4 类角色：项目管理员、资源管理员、系统管理员和备份数据系统。

2. 确定用例

用例是对系统的用户需求（主要是功能需求）的描述，用例描述了系统的功能和所提供的服务。回答 5.2.3 节中的问题，确定本系统用例是：项目管理、资源管理和系统管理。

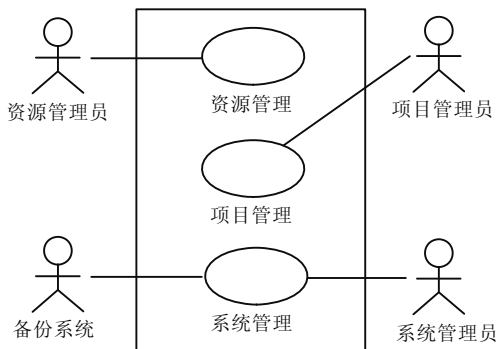


图 5-20 PRMS 高层用例图

3. 画出用例模型概图

用例图是系统的一个功能模型，在绘制用例图时，需要认真考虑它的粒度和抽象层次。按照抽象层次，用例图可以划分为系统层（最高层）、子系统层和对象类层（最低层）。系统高层的用例图也称为用例模型概图，描述了系统的全部功能和服务。图 5-20 所示为 PRMS 高层用例图，描述了该系统的一个最基本的模型。

4. 分解高层的用例图

对高层的用例图进行分解，进一步描述其子系统的功能及服务，并将执行者分配到各层次的用例图中，子系统层又可以自顶而下不断精化，抽象出不同层次的用例图。图 5-21 所示为资源管理子系统的用例图，图 5-22、图 5-23 分别是项目管理用例图和系统管理的用例图。这里需要说明的是，所谓“技能”实际上是指人力资源，而“资源”则是指软、硬件资源。

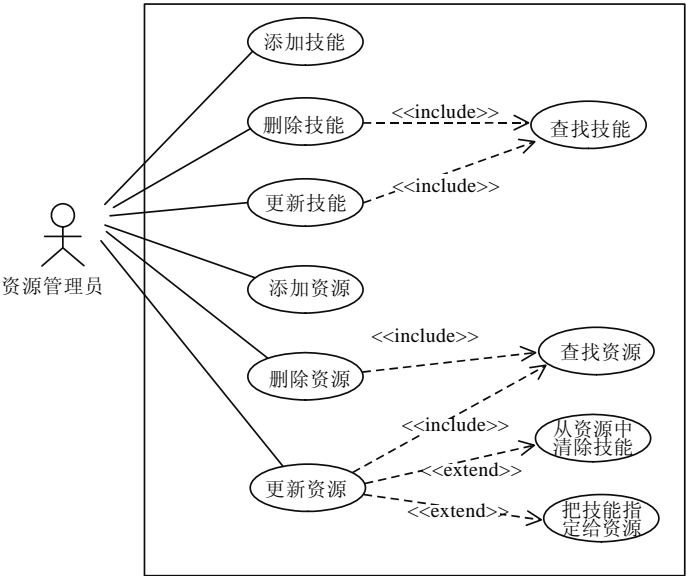


图 5-21 资源管理子系统的用例图

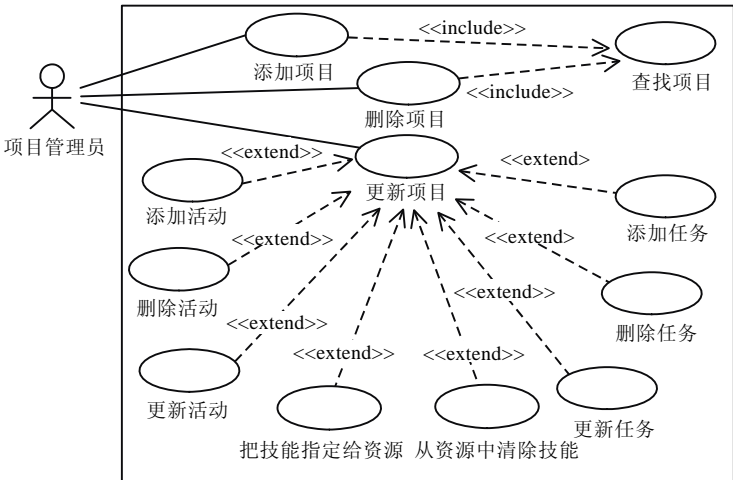


图 5-22 项目管理用例图



请读者分析一下，图 5-21 和图 5-22 中，各用例之间关系<<include>>和<<extend>>的区别。

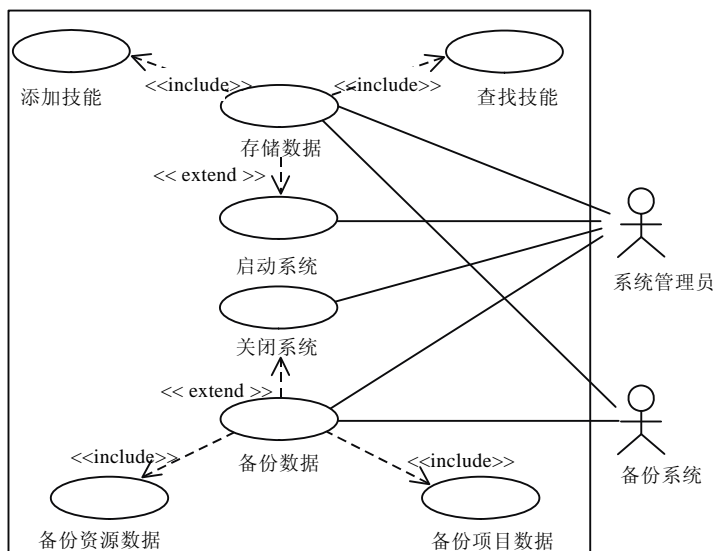


图 5-23 系统管理用例图



5.3 建立静态模型

任何建模语言都以静态建模机制为基础，标准建模语言 UML 也不例外。所谓静态建模，是指对象之间通过属性互相联系，而这些关系不随时间改变而转移。

类和对象的建模，是 UML 建模的基础。熟练掌握基本概念，区分不同抽象层次，以及在实践中灵活运用，是三条最值得注意的建模基本原则。

UML 的静态建模机制包括：用例图（Use Case Diagram）、类图（Class Diagram）、对象图（Object Diagram）、包图（Package Diagram）、构件图（Component Diagram）和配置图（Deployment Diagram）。

其中，用例图已在 5.2 节中讨论过，构件图和配置图将在 5.5 节中进行讨论。由于对象是类的实例（Instance），因此，对象图是类图的变体，两者之间的差别在于，对象图表示的是类图的一个实例。因此，本节主要讨论类图和包图。



5.3.1 类图

类是所有面向对象的开发方法中最重要的基本概念，它是面向对象的开发方法的基础，可以说，UML 的基本任务就要识别系统所必需的类，分析类之间的联系，并以此为基础建立系统的其他模型。类图（Class Diagram）是描述系统静态特征的一种图式，是构建其他图的基础，因此，类图是面向对象技术的核心和灵魂。图 5-24 所示为一个图书管理系统的类图。

构成类图的主要成分是类及类之间的关系。类的图式分为短式和长式两种，短式只标识类名，长式则是对类的完整描述，如图 5-25 所示，由类名、属性和操作三部分构成。属性

描述类的特征。操作也称为方法（或运算），描述该类所能完成的工作，通常以函数的形式出现。在分析阶段，可以先不考虑操作的具体描述，而在设计阶段的类图中导出。

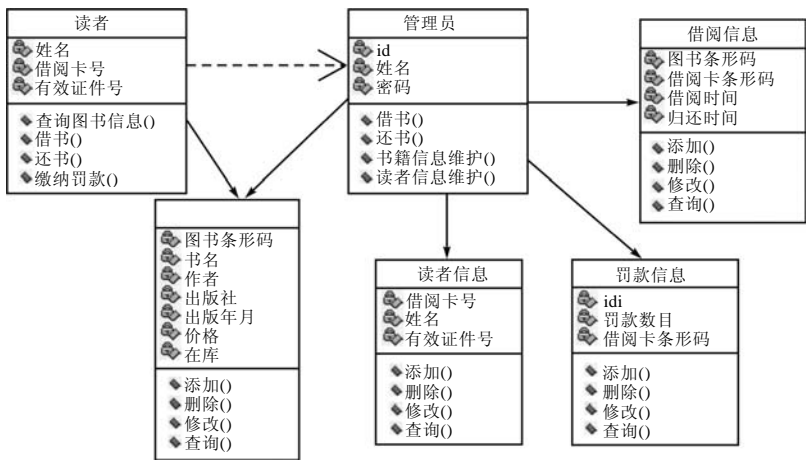


图 5-24 图书管理系统的类图

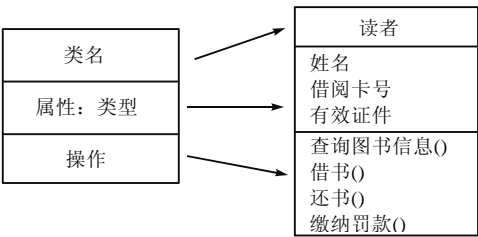


图 5-25 类的图式

为类建模是一件重要而困难的事，因为并不存在一个固定的模式和确定的过程，即使对于同一应用领域项目，不同的系统分析员所得到的类及其属性的集合也可能不同，应该以用户的满意度为标准。为类建模的过程是一个高度迭代增量式的过程。以下内容对于成功设计类是至关重要的：

- ◎ 掌握为类建模的知识；
- ◎ 对应用领域的正确、深入的理解；
- ◎ 研究相似的和成功的设计经验；
- ◎ 超前思维和预测结果的能力；
- ◎ 不断精化模型和修正缺陷的实践。

1. 类的识别

在分析阶段，类的识别通常由分析员在分析问题域的基础上完成。类的识别是面向对象方法的一个难点，但又是建模的关键。常用的方法如下。

(1) 名词识别法

该方法的关键是识别问题域中用名词或者名词短语来描述的实体，通过对系统简要描述进行分析，在提出实体对应名词的基础上识别类。



名词识别法的步骤如下。

① 按照指定语言，对系统进行描述。描述过程应与领域专家共同合作完成，并遵循问题域中的概念和命名。

② 从系统中标识出名词、代词、名词短语，并以此标识为初始的类。

③ 识别、确定（取、舍）类。并非所有列出的名词、代词、名词短语都是类，应根据一定的原则进行识别、确定。

【例 5-5】确定银行网络系统 ATM（Auto Trade Machine）的类。

① 系统简要描述

银行网络系统包括人工出纳和分行共享的自动出纳机；各分理处用自己的计算机处理业务（保存账户、处理事务等）；各分理处与出纳站通过网络通信；出纳站录入账户和事务数据；自动出纳机与分行计算机通信；自动出纳机与用户接口，接受现金卡；发放现金；打印收据；分行计算机与拨款分理处结账。系统应具有记录保管功能和安全措施，要求系统正确处理同一账户的并发访问，每个分理处为自己的计算机准备软件，网络费用平均摊派给各分理处。

② 类的识别

采用名词识别法，检查问题陈述中的所有名词，得到以下初始类：

软件	银行网络	分行计算机	系统	分行	出纳站
出纳员	分理处	分理处计算机	自动出纳机	账户	账户数据
现金卡	事务	事务数据	用户	顾客	现金
收据	访问	费用	安全措施	记录保管	

③ 根据下述原则进一步确定类

◎ 去掉冗余类，若两个类表述同一信息，则应保留最具有描述能力的类，如“用户”与“顾客”是重复的描述，由于“顾客”更具有描述性，故保留它，删除“用户”；去掉不相干的类，删除与问题无关或关系不大的类，如“费用”；删除模糊的类，有些初始类边界定义不确切或者范围太广，应该删除，如“系统”、“安全措施”、“记录保管”、“银行网络”。

◎ 删除那些性质独立性不强，应该是类“属性”的候选类，如“账户数据”、“收据”、“现金”、“事务数据”；删除那些描述的是操作，但不适宜作为对象类，并被其自身所操纵的初始类，如“软件”、“访问”所描述的只是实现过程中的暂时对象，应删去。

通过上述分析，最后确定系统的类为：

分行计算机	分行	出纳站	出纳员	分理处	分理处计算机
自动出纳机	账户	现金卡	事务	顾客	

(2) 实体识别法

该方法与名词识别法很相似，但其关心的是构成系统的实体，而不关心系统的运作流程及实体之间的通信状态。常将实体识别法与名词识别法联合使用。

被标识的实体通常有：系统需要存储、分析、处理的信息实体、系统内部需要处理的设备、与系统交互的外部系统、系统相关人员、系统的组织实体等类型。

系统中的信息实体,往往可以直接识别为类,如职工工资、税务信息等;系统需要处理的设备,可以命名一个同名的类来管理系统对该设备的操作,如有设备传感器,可以命名一个传感器类;而将与本系统交互的外部系统看做一个黑盒,不关心其内部细节,只关心本系统与外部系统的交互情况,可以将其抽象为类。

下面举例说明系统实体识别法的应用。

【例 5-6】有一个购物超市,顾客可在货架上自由挑选商品,由收款机收款,收款机读取商品上的条形码标签,并计算商品价格。收款机应保留所有交易的记录,以供账务复查及汇总使用。

通过分析问题的陈述,确定以下几类实体。

- ① 信息实体:交易记录、商品、税务信息、销售记录、存货记录。
- ② 设备:收款机、条形码扫描器。
- ③ 交互系统:信用卡付款系统。
- ④ 人员职责:收款员、顾客、会计、经理。
- ⑤ 系统的组织实体:本例不考虑。

以上列出的实体,都可以直接识别为类。

(3) 从用例中识别类

用例图是对系统功能的描述,可以根据用例的描述来识别类。该方法与实体识别法很相似,但实体识别法是针对整个系统考虑的,而用例识别法则分别对每一个用例进行识别,因此,用例识别法可能会识别出使用实体识别法未识别出来的类。

针对每个用例,可通过回答以下问题来识别类。

- ① 在用例描述中出现了哪些实体?或者用例的完成需要哪些实体的合作?
- ② 用例在执行过程中会产生并存储了哪些信息?
- ③ 用例要求与之关联的角色应该向该用例输入什么信息?
- ④ 用例向与之关联的角色输出什么信息?
- ⑤ 用例需要对哪些硬设备进行操作?

(4) 利用分解与抽象技术

无论使用哪种方法分析确定类,常使用分解与抽象这两种技术。

① 分解技术

通过以上方法所得到的问题域的类,有的“小类”可能被包含在“大类”之中。所谓“大类”常以整体类和组合类的形式出现,所以分解技术是指对整体类和组合类进行分解的技术。通过分析,对已标识出来的“大类”进行分解,得到新的类,可控制单个类的规模。

在使用分解技术时一定要注意,分解出来的类一定要是系统所需要的相关的类,否则,分解就没有意义。

② 抽象技术

在所识别的类中,可能存在着一些具有相似性的类。所谓相似性是指在信息和动作上的相似性。例如,“汽车”类与“摩托车”类之间的相似性是都有“发动机”。

根据这些类的相似性建立抽象类,并建立抽象类与这些类之间的继承关系。例如,可以建立抽象类“机动车”类,而“汽车”类与“摩托车”类都是通过继承关系而得到



的子类。

抽象类实现了系统内部的重用，很好地控制了复杂性，并为所有子类定义了一个公共的界面，使设计局部化，提高系统的可修改性和可维护性。

使用抽象技术时也要注意，当两个类之间的相似性不强时，要慎重考虑是否需要建立抽象类。掌握抽象技术的难度较大，需要较多的实践经验。

总之，类的识别是 UML 建模的基础和关键，但又比较难于掌握，只有通过对实际系统的分析和设计，逐步加深理解。

2. 类属性与操作识别

(1) 属性 (attribute)

属性用来描述类的特征，表示需要处理的数据。

属性定义：

visibility attribute-name : type = initial-value {property-string}

即

可见性 属性名：类型=默认值{约束特性}

其中，可见性 (visibility) 表示类外的元素是否可访问该属性。可见性分为：

◎ public，公有的，即模型中的任何类都可以访问该属性，用“+”号表示；

◎ private，私有的，表示不能被别的类访问，用“-”号表示；

◎ protected，受保护的，表示该属性只能被该类及其子类访问，用“#”号表示。

如果可见性未申明，则表示其可见性不确定。

(2) 操作 (operate)

对数据的具体处理方法的描述放在操作部分。操作说明了该类能做什么工作。操作通常称为函数，它是类的一个组成部分，只能作用于该类的对象上。

操作定义：

visibility operating-name (parameter-list) : return-type {property string}

即

可见性 操作名 (参数表)：返回类型{约束特性}

其中，可见性同属性中的含义。

参数表为：

参数名：类型，即 parameter-name :type =default-value

返回类型为操作返回的结果类型。

3. 建立类之间的关系

在 UML 中，类之间的关系通常为关联 (Association)、聚合 (Aggregation)、泛化 (Generalization) 和依赖 (Depending)。

(1) 关联

关联是两个或多个类之间的一种关系，链 (link) 是关联的具体体现。关联分为以下 6 种。

① 常规关联

常见的关联是连接类之间的一条直线段，线段上标注关联的名字，可用实心的三角形表

示关联名所指的方向。图 5-26 描述了“公司”类和“员工”类之间的雇用关系。此外，还用重数来描述这两个类之间连接的数量关系。

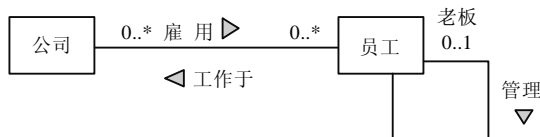


图 5-26 雇用关系

重数的表示通常为：

- 表示“多个”，表示 0 或多个；
- 表示“可选”，表示 0 或者 1 个。

也可在连线上标注数来表示重数：

- “1” ——表示只有 1 个；
- “1+” ——表示 1 个或多个；
- “0..*” 或者 “*” ——表示 0 或者多个；
- “1..*” ——表示 1 或者多个；
- “3..5” ——表示 3~5 个之间整数；
- “2, 4, 15” ——表示 2 个，4 个或 15 个。

重数的默认值为 1。

此外，UML 中还允许一个类与自身关联，称为递归关联（Recursive Association），如图 5-26 中的员工。图 5-27 是对递归关联的一般描述，图 5-28 则描述了具有职责的递归关联，即医生对病人进行治疗。

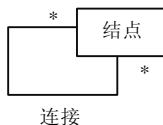


图 5-27 递归关联

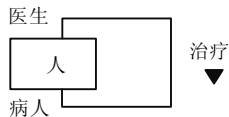


图 5-28 带有职责的递归关联

② 多元关联

关联有二元关联（Binary）、三元关联（Ternary）及多元关联（Higher Order）。两个类之间的关联称为二元关联（见图 5-29），三个类之间的关联称为三元关联（见图 5-30）。对多元关联的描述也可用重数及角色描述。多元关联之间用菱形连接。



图 5-29 二元关联

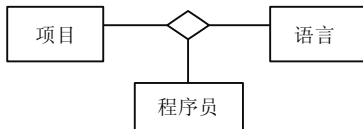


图 5-30 三元关联

图 5-31 描述了一个人（Person）与嗜好（Hobby）的关联，一个人可以有多种嗜好。图 5-32 所示为一个具有重数的三元关联，重数 2..0 表示每个人（Person）在指定的年度



(Year), 最多可以参加两个委员会 (Committee); 而重数 5..3 则表示每个委员会由 3~5 个委员组成; 重数 4..1 表示在一个委员会中, 一个人的任期不超过 4 年。

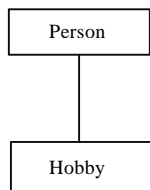


图 5-31 人与嗜好的关联

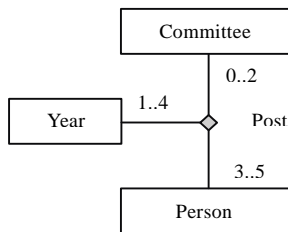
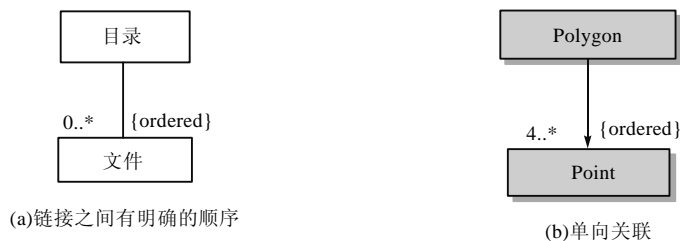


图 5-32 具有重数的三元关联的关联

③ 有序关联

有序是对关联的一种约束, 在关联的“多”端标注{ordered}, 表明这些对象是有序的 (见图 5-33)。关联可用箭头表示该关联使用的方向 (单向或双向), 又称为导引或导航 (Navigation)。

图 5-33 (a) 描述了一个目录下可以有多个有序的文件, 而一个文件只属于一个目录。图 5-33 (b) 表示多边形的多个顶点的有序关联。



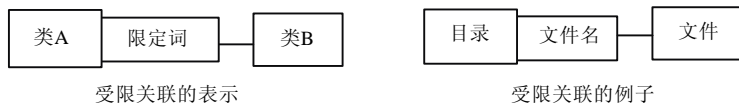
(a) 链接之间有明确的顺序

(b) 单向关联

图 5-33 有序关联

④ 受限关联 (Qualified Association)

如果对关联的含义做出某种限制, 则称为受限关联。使用限定词对该关联一端的对象进行明确的标识和鉴别。图 5-34 中, 文件名是限定词, 一个目录中有多个文件, 加上文件名这个限定词, 确定了文件的唯一性。



受限关联的表示

受限关联的例子

图 5-34 受限关联

⑤ 或关联

图 5-35 描述了一个签订保险合同的类图, 由于人和公司不能拥有同一份合同, 因此描述为或的关系, 用虚线连接两个关联并标注{or}。

⑥ 关联类

在 UML 中, 当仅用关联名不足以描述关联, 需要对其进行更详细的说明时, 可以将关联

定义为类。如图 5-36 所示，为了对关联做进一步描述，定义“授权”关联类及其属性和操作。

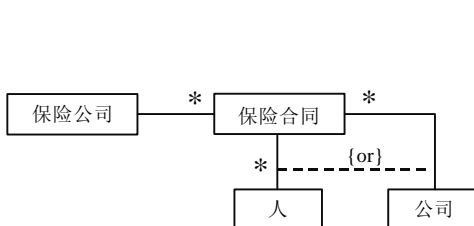


图 5-35 或关联

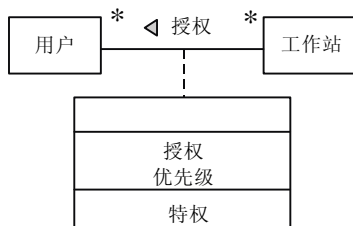


图 5-36 关联类

(2) 聚合（Aggregation）关系

聚合是一种特殊的关联，它指出类间的“整体-部分”关系，又分为共享聚合和组合聚合。

① 共享聚合（Shared Aggregation）

其“部分”对象可以是任意“整体”对象的一部分。当“整体”端的重数不是 1 时，称聚合是共享的。在“整体”端用一个空心小菱形来表示共享聚合，如图 5-37 所示。

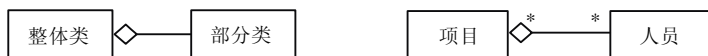


图 5-37 共享聚集

② 组合聚合（Composition Aggregation）

组合聚合的“整体”与“部分”之间的关系较共享聚合更加紧密。其“整体”（重数为 0, 1）拥有它的“部分”，部分仅属于同一整体，或者说整体与部分必须同时存在，如图 5-38 所示。图 5-39 描述了组合聚合的三种等价形式。



图 5-38 组合聚合

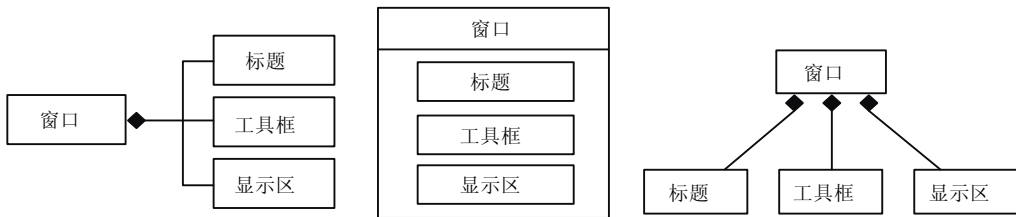


图 5-39 组合聚合的三种等价形式

(3) 泛化关系

在 UML 中，泛化关系指出类之间的一般与特殊关系，它是通用元素与具体元素之间的一种分类关系，通常表现为继承关系（见图 5-40）。一般类即父类，通过特化得到子类。泛化关系用一个三角形表示，三角形的尖端对着一般类。

父类与子类之间可构成类的分层结构，如图 5-41 所示为一个分层继承类图的实例。图中一般类为“图形”，按照维数泛化为“0 维”、“1 维”、“2 维”，构成一棵继承树。其中，“图



形”类及“1 维”类、“2 维”类为抽象类。

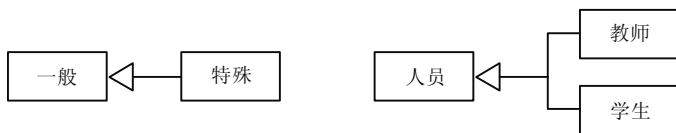


图 5-40 泛化

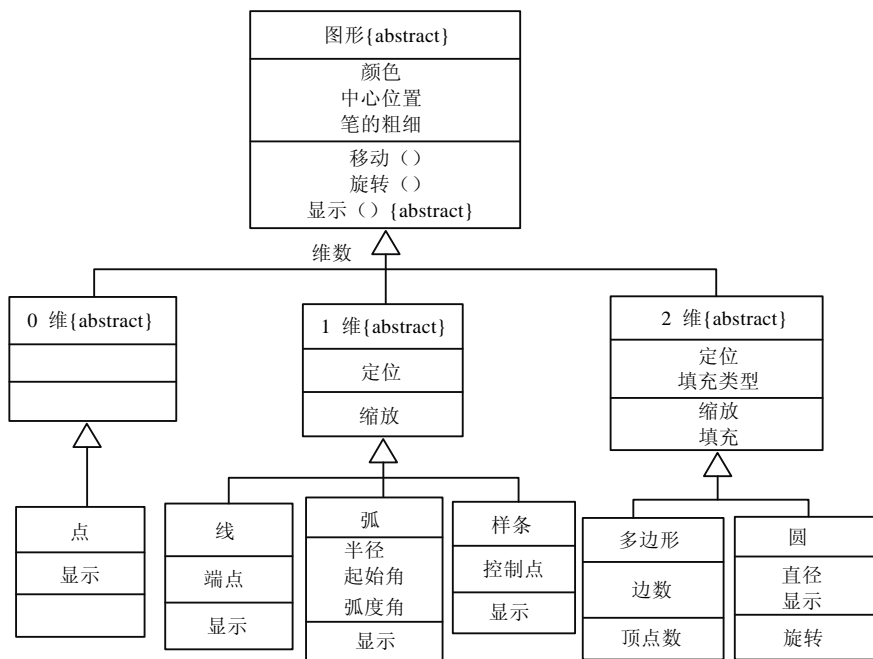


图 5-41 泛化关系

抽象类：指没有实例的类，它定义一些抽象的操作，即不提供实现方法的操作，只提供操作的特征，并标注{abstract}。

还有几种特殊的泛化关系。

重叠泛化：在继承树中，若存在某种具有公共父类的多重继承，则称为是交叠的，并标注{overlapping}，如图 5-42 中的“水陆两栖车”类，否则是不交的{disjoint}。

完全泛化：一般类特化出它所有的子类，称为完全泛化，标记为{complete}（见图 5-43）。

不完全泛化：一般类未特化出它所有的子类，称为不完全泛化，标记为{incomplete}。

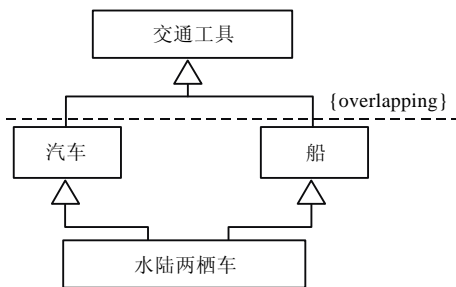


图 5-42 重叠泛化

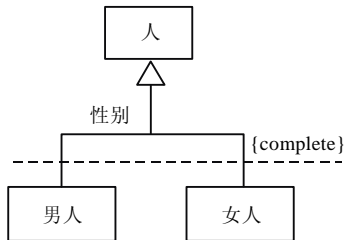


图 5-43 完全泛化

图 5-44 描述了一个 Person 对象与交通工具之间有关联“驾驶”，当 Person 对象使用交通工具的 drive 操作时，具体结果将取决于所操作的对象。如果是汽车对象，则 drive 对应启动轮子转动；如果是轮船对象，则 drive 对应启动螺旋桨。这种在子类中重新定义父类的某些操作的技术，称为多态性。

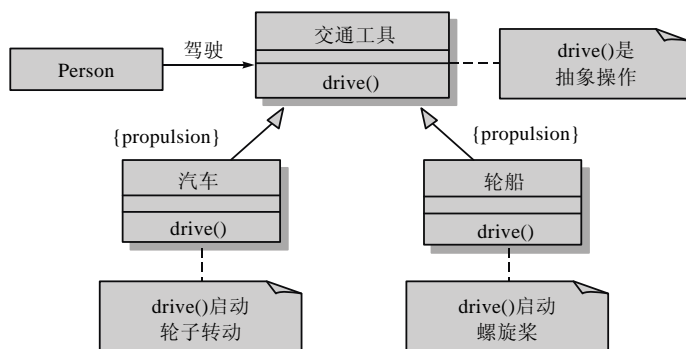


图 5-44 泛化中的多态

此外，在泛化关系中还可采用识别名称（Discriminator）来指明泛化中一般化到具体化的主要依据。因此，在交通工具与汽车和轮船的泛化关系中，识别名称为驱动方式（Propulsion）。

图 5-45 是一个关于订单的类图，其中，订单类（Order）与订单行类（OrderLine）之间有 1 对多的关联，角色 LineItem（行项目）表示订单行是订单的一个行项目。

客户类（Customer）与团体客户类（Corporate Customer）和个人客户类（Personal Customer）之间是继承关系。图 5-45 中还描述了职员类（Employee）、产品类（Product）与其他类之间的关联。

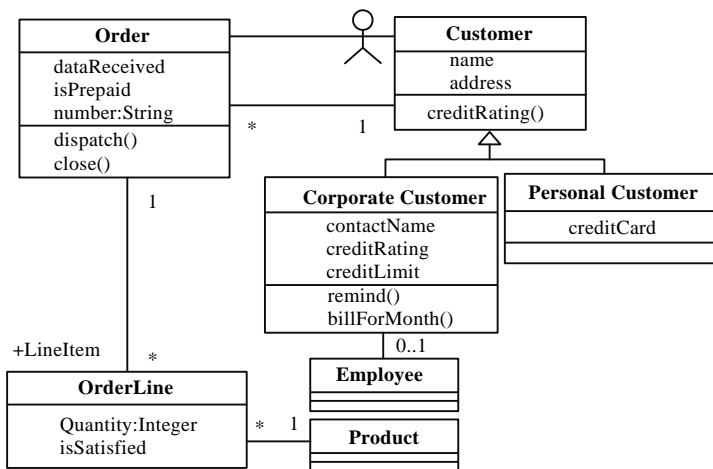


图 5-45 类图

4. 类图的抽象层次和细化（Refinement）关系

需要注意的是，虽然在软件开发的阶段都使用类图，但这些类图表示了不同层次的抽象。在需求分析阶段，类图是研究领域的概念；在设计阶段，类图重点描述类与类之间的



接口；而在实现阶段，类图描述软件系统中类的实现。

按照 Steve Cook 和 John Dianiels 的观点，类图分为三个层次：概念层（Conceptual）、说明层（Specification）和实现层（Implementation）。

概念层的类图描述应用领域中的概念。实现它们的类可以从这些概念中得出，但两者并没有直接的映射关系。事实上，一个概念模型应独立于实现它的软件和程序设计语言。

说明层的类图描述软件的接口部分，而不是软件的实现部分。面向对象开发方法非常重视区别接口与实现之间的差异，但在实际应用中却常常忽略这一差异，这主要是因为 OO 语言中类的概念将接口与实现合在了一起。大多数方法由于受到语言的影响，也仿效了这一做法。现在这种情况正在发生变化，可以用一个类型（Type）描述一个接口，这个接口可能因为实现环境、运行特性或者用户的不同而具有多种实现方式。实现层才真正有类的概念，并描述软件的实现部分，这可能是大多数人最常用的类图。但在很多时候，说明层的类图更易于开发者之间的相互理解和交流。理解以上层次，对于画类图和读懂类图都是至关重要的。但由于各层次类图之间没有一个清晰的界限，所以大多数建模者在画图时没能对其加以区分。画图时，要从一个清晰的层次观念出发；而读图时，则要弄清它是根据哪种层次观念来绘制的。

需要说明的是，这个观点同样也适合于其他任何模型，只是在类图中显得更为突出，更好地描述了类图的抽象层次和细化（Refinement）关系。

5.3.2 包图

将许多类集成一个更高层次的单位，形成一个高内聚、低耦合的类的集合，UML 中把这种分组机制称为包。包（Package）是一种分组机制，包由关系密切的一组模型元素构成，包还可以由其他包嵌套构成。引入包是为了降低系统的复杂性，包图是维护和控制系统总体结构的重要建模工具。包的表示如图 5-46（a）所示。

构成包的模型元素称为包的内容，包通常用于对模型的组织管理，因此有时又将包称为子系统（Subsystem）。包拥有自己的模型元素，包与包之间不能公用一个相同的模型元素，包的实例没有任何语义（含义），仅在模型执行期间包才有意义。

包的内容可以是类的列表，也可以是另一个包图，还可以是一个类图。包之间的关系有依赖和泛化（继承）。

依赖关系：如果两个包中的任意两个类存在依赖关系，则称为包之间存在依赖关系。包之间的依赖关系，最常用的是输入依赖关系<<import>>与<<access>>，两者之间的区别是，后者不把目标包内容加到源包的名字空间中。图 5-46（b）表示了包及包之间的依赖关系。

泛化关系：使用继承中通用和特例的概念来说明通用包和专用包之间的关系。例如，专用包必须符合通用包的界面，与类继承关系类似，如图 5-46（c）所示。

和类一样，包也有可见性，利用可见性来控制外部包对包的内容的存取方式。UML 中定义了 4 种可见性：私有、公有、保护和实现。包的可见性默认为公有的。图 5-47 中，引入（import）是依赖关系的变种，其含义是：允许一个包接受或者访问另一个包中的公共内容。

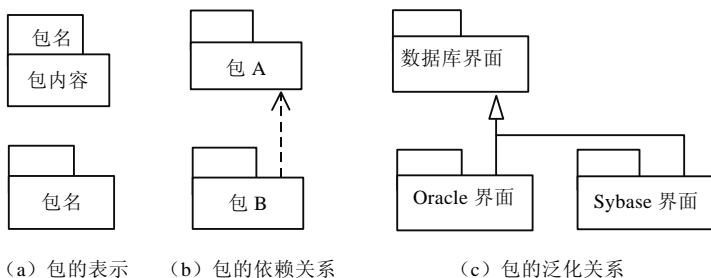


图 5-46 包

包也可以有接口，接口用实线连接的小圆圈表示，接口通常由包的一个或多个类实现，如图 5-48 所示，包 A 的接口为 I。包 X 通过接口 I 对包 A 有依赖关系。

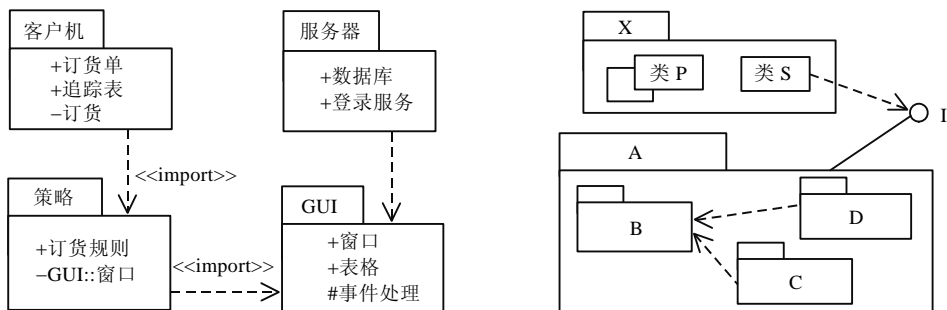


图 5-47 包之间的依赖关系

图 5-48 包之间的接口



5.4 建立动态模型

动态模型主要用于描述系统的动态行为和控制结构。动态行为包括系统对象生命周期内可能的状态、事件发生时状态的转移及对象之间的动态合作关系，显示对象之间的交互过程及交互顺序，同时描述了为满足用例要求所进行的活动及活动之间的约束关系。

动态模型包括 4 类图。

- ① 状态图（State Diagram）：用来描述对象、子系统、系统的生命周期。
- ② 活动图（Activity Diagram）：着重描述操作实现中完成的工作及用例或对象的活动，活动图是状态图的一个变种。
- ③ 顺序图（Sequence Diagram）：是一种交互图，主要描述对象之间的动态合作关系以及合作过程中的行为次序，常用来描述一个用例的行为。
- ④ 合作图（Collaboration Diagram）：用于描述相互合作的对象间的交互关系，它描述的交互关系是对象间的消息连接关系。



5.4.1 消息

在动态模型中，对象间的交互是通过对象间消息的传递来完成的。对象通过相互间的消



息传递（通信）进行合作，并在其生命周期内根据通信的结果不断改变自身的状态。在 UML 中，消息的图形表示是，用带有箭头的线段将消息的发送者和接收者联系起来（见图 5-49），箭头的类型表示消息的类型。

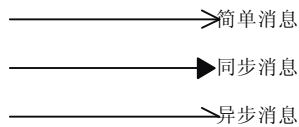


图 5-49 消息的类型

① 简单消息（Simple）：表示简单的控制流，描述控制如何从一个对象传递到另一个对象，但不描述通信的细节。

② 同步消息（Synchronous）：是一种嵌套的控制流，用操作调用实现，操作的调用是一种典型的同步消息。操作的执行者要等到消息相应操作执行完并回送一个简单消息后，再继续执行。

③ 异步消息（Asynchronous）：表示异步控制流，消息的发送者在消息发送后，不用等待消息的处理和返回即可继续执行。异步消息主要用于描述实时系统中的并发行为。

5.4.2 状态图

状态图（State Diagram）用来描述一个特定对象的所有可能的状态及其引起状态转移的事件。一个状态图包括一系列的状态以及状态之间的改变。

1. 状态

一个对象在其生命期中都具有多个状态。通常，状态具有时间稳定性。状态是对象执行了一系列活动的结果。当某个事件发生后，对象的状态将发生变化。状态图中定义的状态如下。

初态——状态图的起始点。一个状态图只能有一个初态。

终态——是状态图的终点。终态可以有多个。

中间态——可包括三个区域：名字域、状态变量与活动域。

复合态——可以进一步细化的状态。

对象的状态如图 5-50 所示。

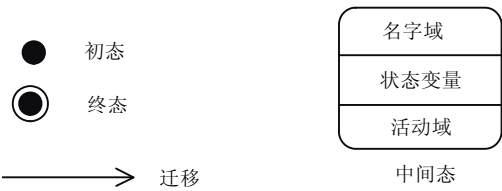


图 5-50 对象的状态

在中间态中，名字域即为状态名；状态变量表示状态图所显示的类的属性；活动域则列出了在该状态时要执行的事件和动作，即响应事件的内部动作或活动的列表，定义为：

事件名（参数表[条件]）/动作表达式

UML 预设了 3 个标准事件，而且都是无参数事件：

- ① entry 事件，用于指明进入该状态时的特定动作；
- ② exit 事件，用于指明退出该状态时的特定动作；
- ③ do 事件，用于指明在该状态时执行的动作。

图 5-51 描述了 login 状态。

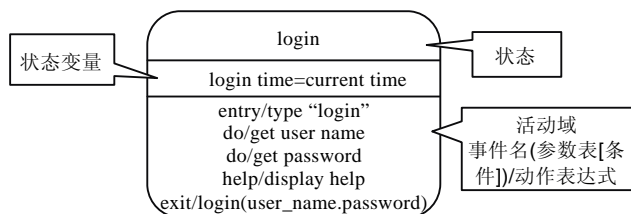


图 5-51 login 状态

2. 状态迁移

一个对象的状态改变称为状态迁移，通常是由事件触发的。事件是激发状态迁移的条件或操作。在 UML 中，有以下 4 类事件：

- ① 某条件变为真，表示状态迁移上的警戒条件；
- ② 收到来自外部对象的信号（Signal），表示为状态迁移上的事件特征，也称为消息；
- ③ 收到来自外部对象的某个操作中的一个调用，表示为状态迁移上的事件特征，也称为消息；
- ④ 状态迁移上的时间表达式。

在状态图中，一般应标出触发转移的事件表达式。如果转移上未标明事件，则表示在源状态的内部活动执行完毕后自动触发转移。

3. 画状态图的步骤

- ① 确定要描述的对象（不是全部），选择那些状态变化对实现系统功能影响大的对象。
- ② 确定状态空间，即对象在其生命期中所有状态的总和，包括确定状态的粒度。粒度反映了分析者对问题域的理解和问题域的本质。
- ③ 确定引起状态迁移的事件。

【例 5-7】画出电梯升降的状态图。

解：首先分析电梯移动的状态空间，确定电梯升降有 5 个状态，3 个循环圈，分别为：“空闲”状态与“向下移动”状态之间，“空闲”状态与“向上移动”状态之间，以及大循环。循环圈越多，表明对象的控制逻辑越复杂。

确定并标注引起状态转移的事件，“超时”是指电梯的空闲状态超过某个规定的时间值。图 5-52 描述了电梯升降的状态图。

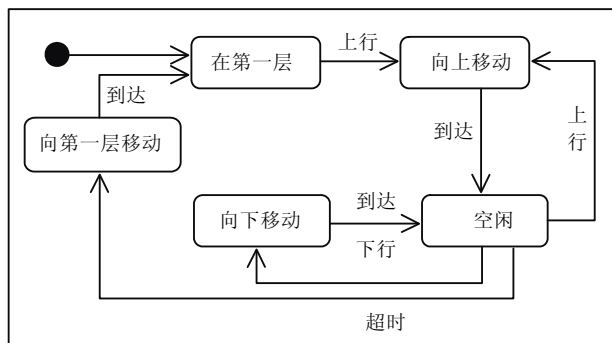


图 5-52 电梯升降状态图



4. 细化的状态图

为了进一步描述对象状态的变化, 可对中间状态进行细化, 图 5-53 给出了电梯升降状态图的细化状态表示。图中对系统中对象状态的状态变量和活动做了进一步的描述, 状态变量为 `timer`, 初值为 0, 活动为 `do/increase timer`。

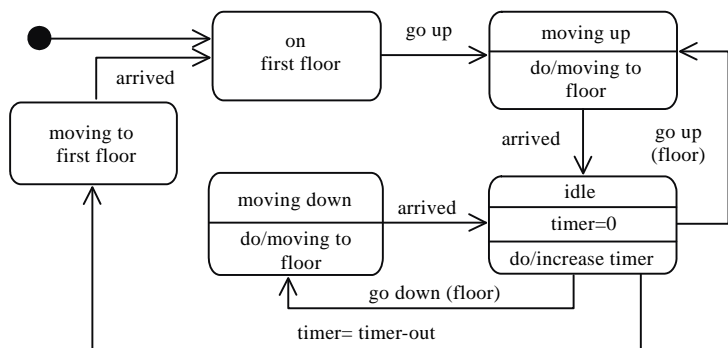


图 5-53 细化电梯状态图

5. 嵌套的状态图

状态图可能有嵌套的子状态图, 且子状态图可以是另一个状态图。子状态又可分为两种: “与”关系的子状态和“或”关系的子状态。图 5-54 中, “行驶”状态有两个“或”关系的子状态——“向前”与“向后”。而图 5-55 中, “前进”与“后退”, “低速”与“高速”分别是两对或关系的子状态, 而虚线上、下又分别构成两个“与”关系的子状态。

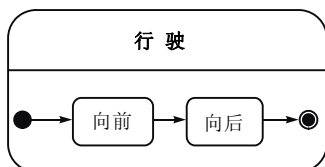


图 5-54 “或”关系的子状态

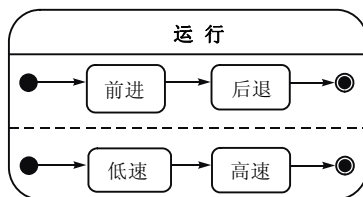


图 5-55 “或”关系的子状态及“与”关系的子状态

6. 状态图之间的消息发送

状态图之间可以发送消息, 用虚箭头表示, 这时, 状态图必须画在矩形框中, 如图 5-56 所示。

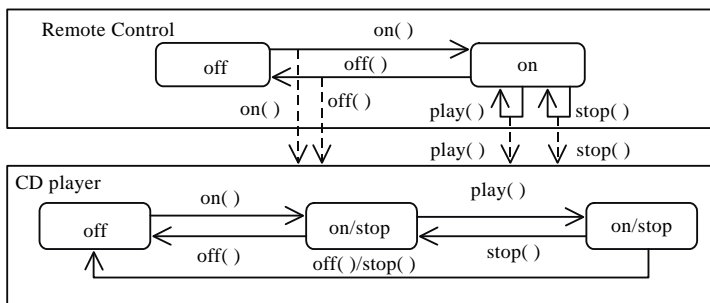


图 5-56 消息发送状态图

图 5-57 描述了银行 ATM 系统用户取款的状态图，包括从用户插卡到取款的所有可能的状态及其变化情况。

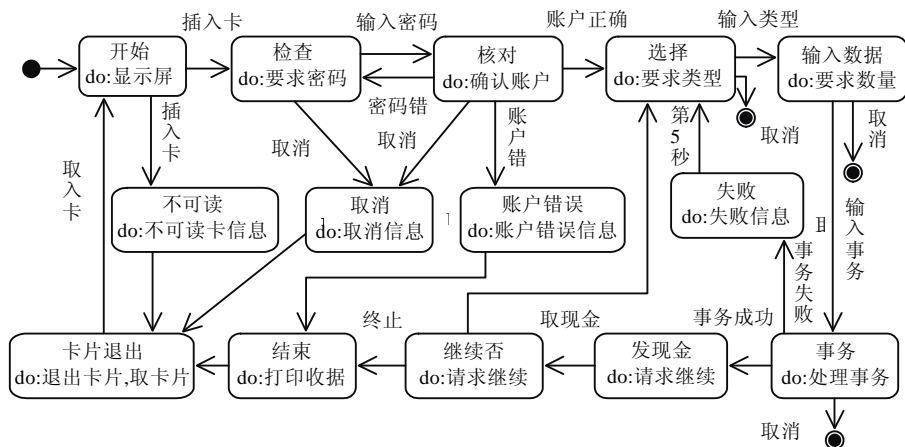


图 5-57 银行 ATM 系统用户取款的状态图

5.4.3 顺序图

顺序图（Sequence Diagram）用来描述一组对象之间动态的交互关系，着重描述对象间消息传递的时间顺序。

1. 概述

顺序图有两个轴：水平轴列出参与交互的不同对象，用标有对象名的矩形框表示，对象名下标注实线以区别于类；垂直虚线是对象的生命线，用于表示在某段时间内对象的存在。

对象间的通信和交互通过在对象的生命线之间传送的消息来表示。消息分为：简单消息（Simple）、同步消息（Synchronous）和异步消息（Asynchronous）。

顺序图中还常常给出消息的说明信息，用于说明消息的名称、序号、发送的时间及动作执行的情况，还可以定义两个消息之间的时间限制及一些约束信息等。

2. 消息的说明信息

消息延迟：用倾斜箭头表示，表示消息执行时间的延迟。

消息串：包括消息和控制信号。控制信息位于信息串的前部。

控制信息 { 条件控制信息，如：[$x > 0$]
重复控制信息，如：* [$I = 1..n$]

当收到消息时，接收对象立即开始执行活动，即对象被激活了，并在对象生命线上显示一个细长的矩形框来表示激活（见图 5-58）。

3. 顺序图的形式

有两种格式的顺序图：一般格式和实例格式。实例格式详细描述一次可能的交互，没有任何条件和分支或循环，它仅仅显示选定情节（场景）的交互（见图 5-58）。而一般格式则描述所有的情节，可能包括了分支、条件和循环。

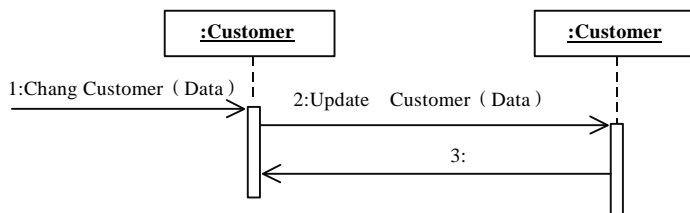


图 5-58 顺序图

图 5-59 描述了带分支和循环的顺序图。

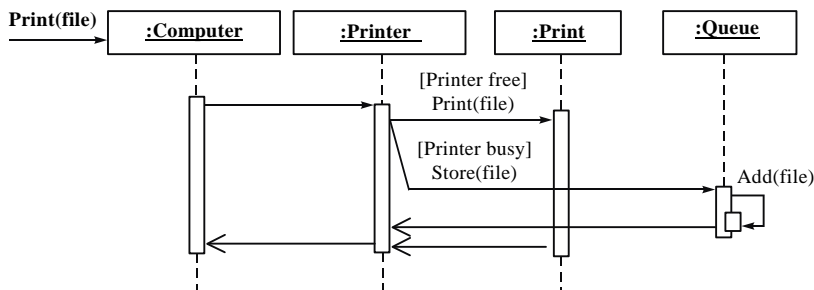


图 5-59 带分支和循环的顺序图

图 5-60 所示为打电话的顺序图。图中，有呼叫者、交换、接收者三个对象，对象之间传送消息，其中路由选择用斜线箭头表示延时。A、B、C、D、E 表示消息发送和接收的时刻，花括号中的信息表示时间限制。这些都是说明信息。

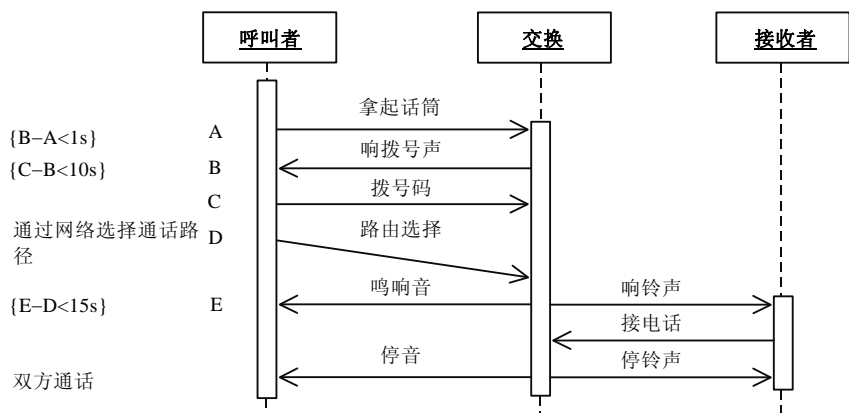


图 5-60 打电话的顺序图

4. 创建对象与对象的消亡

在顺序图中，还可以描述出一个对象通过发送一条消息来创建另一个对象。如图 5-61 所示，对象 Customer 是由对象 Customer Windows 发送消息 Customer (Data) 而创建的。当对象消亡 (Destroying) 时，用符号 “×” 表示。

【例 5-8】有一个对外营业的会议中心，有各种不同规格的会议室，为用户提供召开和组织会议的服务：如预订会议、召开会议、会务管理等。请建立申请召开会议 (Request Meeting Instance) 的顺序图。

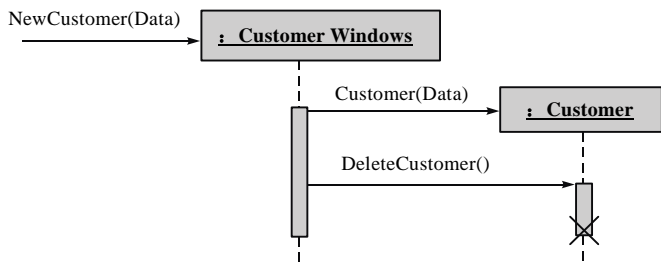


图 5-61 创建或删除对象

用户要召开一个会议，首先要进行申请，并指定会议的名称、召开的时间、会议参加人员，并确定会议室。参与交互的对象有：管理会议（MeetingAdministration）、会议（Meeting）、会议的描述（MeetingInstance）和会议室（MeetingRoom）。其中，MeetingInstance 与 Meeting 之间是继承关系。在图 5-62 中，instance、member、group、room、info 都是临时对象，instance 记录了用户指定的会议属性（时间、参加人数等），member 为一个参会代表，是参会人员组 group 的对象；而 room 是满足要求的会议室。

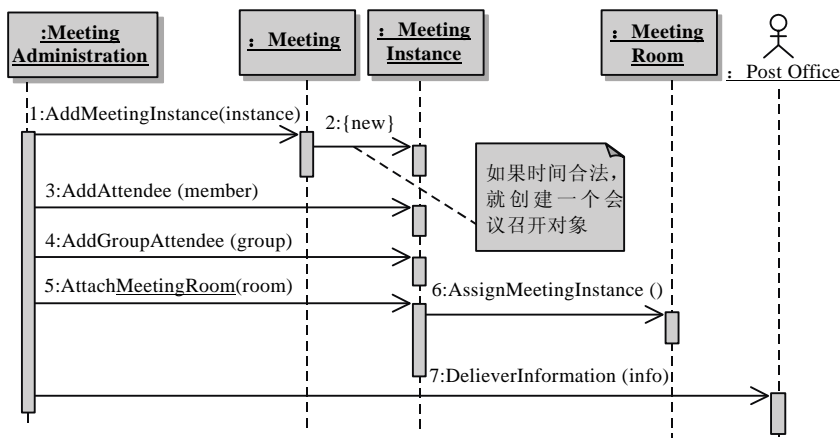


图 5-62 申请召开会议的顺序图



5.4.4 合作图

合作图（Collaboration Diagram），也称为协作图，用于描述相互合作的对象间的交互关系和链接（Link）关系。虽然顺序图也描述对象间的交互关系，但其侧重点不同。顺序图着重体现交互的时间顺序，合作图则着重体现交互对象间的静态链接关系。

例如，图 5-63 所示为一个打印文件的合作图，由 3 个对象：Computer、PrinterServer 和 Printer 合作完成打印文件的工作。当操作者向对象 Computer 发出打印文件的消息后，如果打印机空闲，则 Computer 向 PrinterServer 对象发送“1: Print(ps-file)”，PrinterServer 再向对象 Printer 发送消息“1.1: Print(ps-file)”。

合作图不强调执行事件的顺序，而是强调为了完成某个任务，对象之间通过发送消息实现协同工作关系。

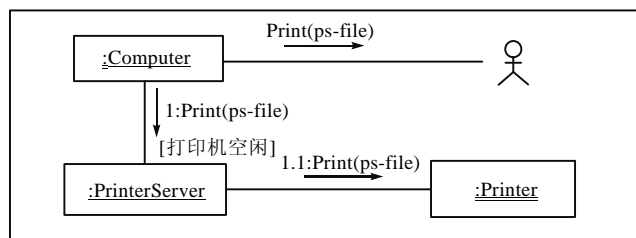


图 5-63 合作图

1. 合作图中的模型元素

(1) 对象

合作图中如果一个对象在消息的交互中被创建，则可在对象名称之后标上{new}。类似地，如果一个对象在交互期间被删除，则可在对象名称之后标上{destroy}。



(2) 链接

链接（Link）用于表示对象间的各种关系，包括组成关系的链接（Composition Link）、聚合关系的链接（Aggregation Link）、限定关系的链接（Qualified Link）以及导航链接（Navigation Link）等。各种链接关系与类图中的定义相同。

(3) 消息

在对象之间的静态链接关系上标注消息。消息类型同样有简单消息、同步消息和异步消息三种。用标号表示消息执行的顺序。消息定义的格式如下：

消息类型 标号 控制信息：返回值：=消息名参数表

其中，标号有 3 种：

- ◎ 顺序执行，按整数大小执行，如 1, 2, 3, ...
- ◎ 嵌套执行，标号中带小数点，如 1.1, 1.2, 1.3, ...
- ◎ 并行执行，标号中带小写字母，如 1.1.1a, 1.1.1b, ...

2. 合作图应用举例

【例 5-9】画出一个电路设计系统进行布线过程的合作图。

首先确定参与布线的对象，对象的确定源于对系统进行分析时所确定的类。本系统确定了控制器、窗口、布线等对象。

对象之间的合作过程为：控制器控制窗口的工作，并在其控制下开始进行布线。每次布线，先要定位两个端点，同时确定左端点 r0 和右端点 r1，再以 r0 和 r1 为参数，创建直线对象，并将其在窗口对象上显示出来。

图 5-64 中的消息是嵌套的标号，表示消息发送的嵌套执行顺序为：首先执行消息 1，重复执行 1.1，并行执行 1.1.1a 和 1.1.1b 以确定左、右端点，再发送消息 1.1.2 创建直线，消息 1.1.3 将创建的直线在窗口对象中显示，消息 1.1.3.1 则在窗口对象中增加一条新布的线。

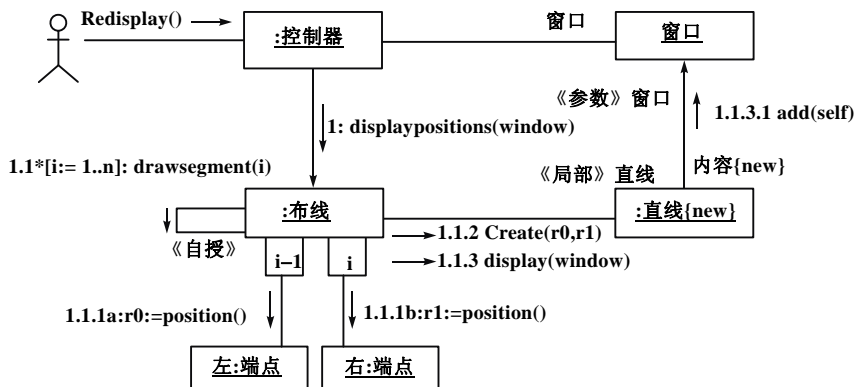


图 5-64 电路设计的合作图

整个布线过程是通过对象之间发送一系列的消息来实现的。在合作图中，虽然消息也按照标号的嵌套顺序执行，但并不像顺序图那样强调消息执行顺序，而主要描述对象之间的协作关系。

【例 5-10】图 5-65 所示为一个统计销售结果的合作图，请读者自己分析对象之间的关系及消息的发送过程。**错误！**

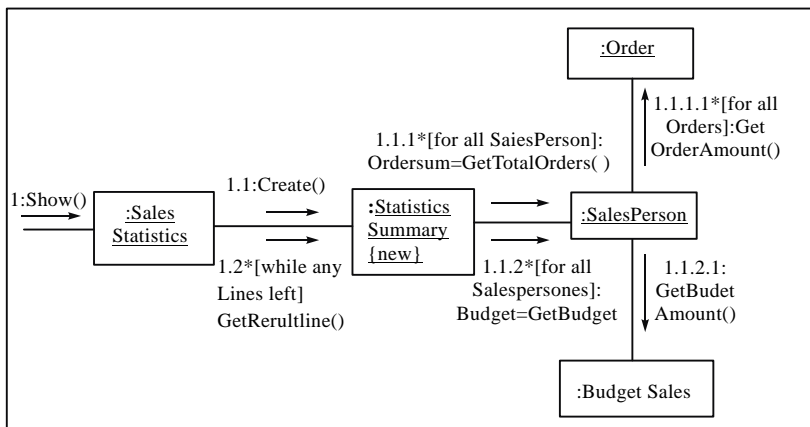


图 5-65 统计销售结果的合作图

5.4.5 活动图

活动图（Activity Diagram）是由状态图变化而来的，它们各自用于不同的目的。前面介绍的三类动态模型，已从不同角度对系统的动态特征进行了描述，状态图着重描述了对象的状态变化以及触发状态变化的事件，交互模型（顺序图和合作图）则描述对象之间的动态交互行为。那么，为什么还要引入活动图呢？有以下两个原因。

① 从系统任务的观点来看，系统的执行过程是由一系列有序活动组成的。活动图可以有效地描述整个系统的流程，即活动图描述的是系统的全局的动态行为。

② 前面介绍的三类动态模型，都无法描述系统任务中大量存在的并发活动，只有活动



图是唯一能够描述并发活动的 UML 图。

活动图还描述了系统中各种活动的执行顺序,刻画一个方法中所要进行的各项活动的执行流程。活动图的应用非常广泛,它既可用来描述操作(类的方法)的行为,也可以描述用例和对象内部的工作过程,并可用于表示并行过程。活动图显示动作及其结果,着重描述操作实现中完成的工作以及用例或对象内部的活动。在状态图中,状态的变迁通常需要事件的触发;而在活动图中,一个活动结束将立即自动进入下一个活动。

1. 活动图的构成

构成活动图的模型元素有:活动、转移、对象、信号、泳道等。其中,活动是活动图的核心概念。

(1) 活动

活动是构成活动图的核心元素,是具有内部动作的状态,由隐含的事件触发活动的转移。活动的解释依赖于作图的目的和抽象层次。在概念层描述中,活动表示要完成的一些任务;在说明层和实现层中,活动表示类中的方法。

活动用圆角矩形框表示,框内标注活动名。活动图的图符如图 5-66 所示。在活动图中使用一个菱形表示判断(Decision),来表达条件关系,是一种特殊的活动。判断标志可以有多个输入和输出转移,但在活动的运作中仅触发其中的一个输出转移。同步也是一种特殊的活动,同步线描述了活动之间的同步关系。

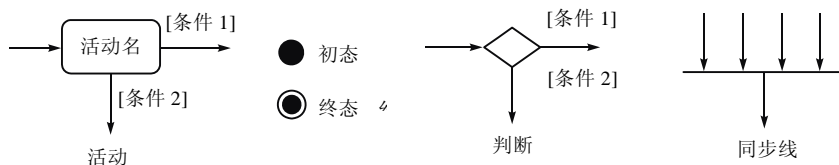


图 5-66 活动图的图符

(2) 转移

转移描述活动之间的关系,描述由于隐含事件引起的活动变迁,即转移可以连接各活动及特殊活动(初态、终态、判断、同步线)。

转移用带箭头的直线表示,可标注执行该转移的条件。若无条件标注,则表示顺序执行。

(3) 泳道

活动图描述要执行的活动和顺序,但并没有描述这些活动是由谁来完成。泳道(SwimLane)进一步描述完成活动的对象,并聚合一组活动,因此泳道也是一种分组机制。

将一张活动图划分为若干个纵向矩形区域,每个矩形区域称为一个泳道,包括了若干活动,在泳道顶部标注的是完成这些活动的对象,图 5-67 描述了一个顾客购物的过程。

活动图中只有一个起点,表示方式与状态图一样,泳道被用来组合活动,通常根据活动的功能来组合。

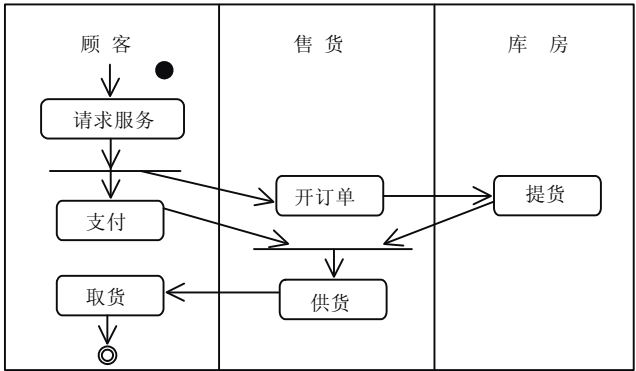


图 5-67 泳道

(4) 对象流

活动图中还可以出现对象。对象作为活动的输入/输出，用虚箭头表示。图 5-68 中，测量活动所产生的结果输出给对象测量值，再由该对象将值传送给显示活动。

(5) 控制图符

活动图中可发送和接收信号，分别用发送和接收图符表示，如图 5-69 所示。发送符号对应于与转移联系在一起的发送短句，接收符号也同转移联系在一起。图 5-70 描述了一个调制咖啡的过程，将“开动”信号发送给对象“咖啡壶”，当调制咖啡完成后，将接收来自对象“咖啡壶”的“信号灯灭”信号。

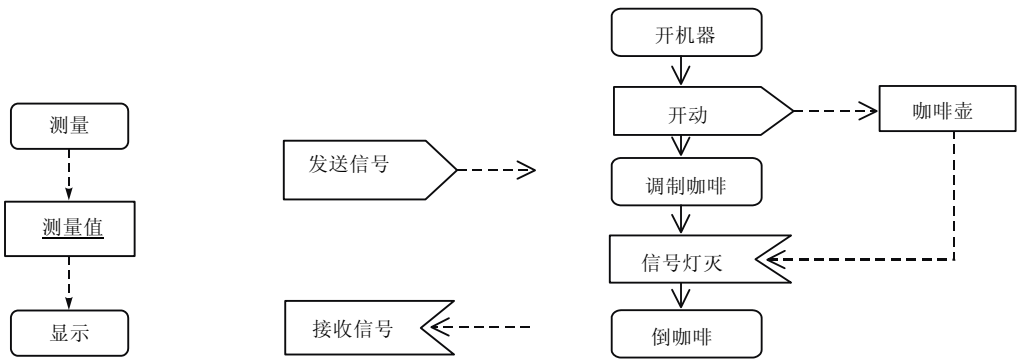


图 5-68 对象流

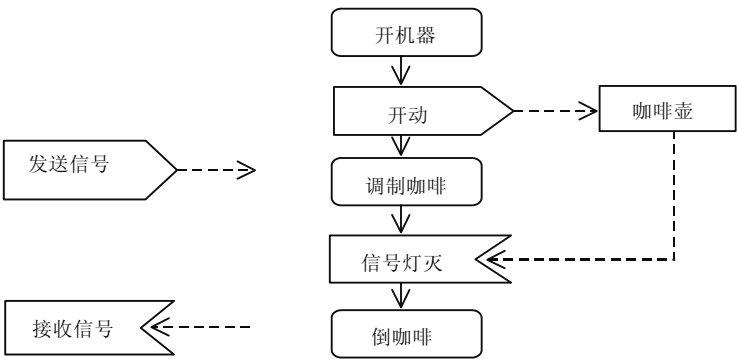


图 5-69 控制图符

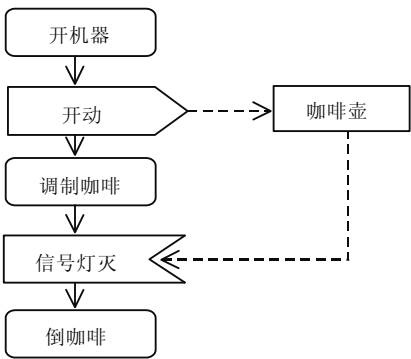


图 5-70 控制图符举例

2. 活动图举例

【例 5-11】建立医院病房监护系统的活动图。

医院病房监护系统，主要活动由“采集病症信号”、“分析比较信号”、“判断病症异常否”、“报警”、“打印病情报告”等活动组成。当病症出现异常时，立即报警，同时打印病情报告和更新病历，三者是并发执行的活动，因此使用同步线描述（见图 5-71）。

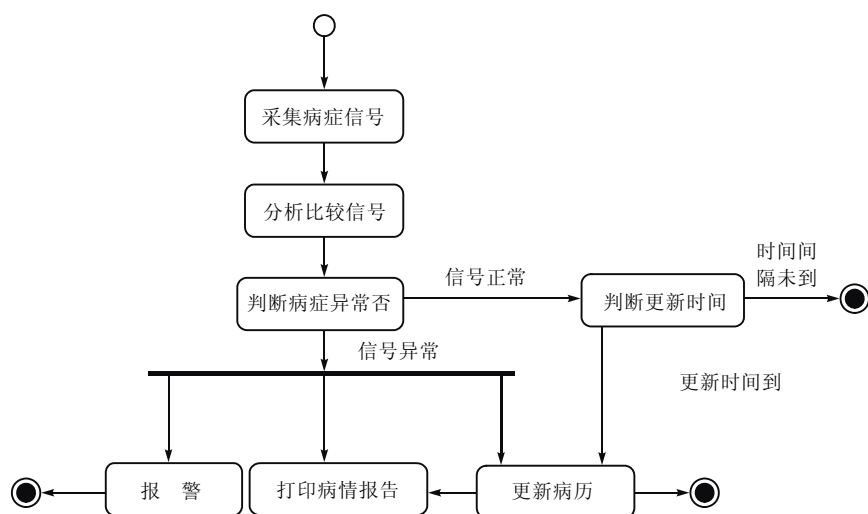


图 5-71 医院病房监护系统的活动图



5.5 建立实现模型

实现模型描述了系统实现时的一些特性，又称为物理体系结构模型，包括源代码的静态结构和运行时刻的实现结构，它由构件图和配置图组成。

5.5.1 构件图

构件图（Component Diagram）又称为组件图，它显示代码本身的逻辑结构，描述系统中存在的软构件以及它们之间的依赖关系。构件图的主要元素有构件、依赖关系和界面。

构件（Component）代表系统的一个物理组件，表示系统本身的物理代码模块。构件在逻辑上与包、类对应，实际上是一个文件。构件名描述构件实现的功能，同时标注对应的文件名。构件可以由类和其他构件组合而成。

1. 构件的类型

① 源代码构件（Source Component），源代码构件是实现一个或者多个类的源代码文件。可在构件上标注以下符号：

《file》——表示包含源代码的文件；

《page》——表示 Web 页；

《document》——表示文档，而不是可编译的代码。

② 二进制构件，它是一个目标代码文件，或通过编译一个或多个源代码构件生成的静态库文件或动态库文件。

③ 可执行构件（Executable Component），即在 CPU 上运行的一个可执行的文件。

2. 构件之间的关系

构件之间的关系主要有两类：依赖关系和接口。

(1) 依赖关系

构件之间的依赖关系用虚线箭头表示，又分为开发期的依赖和调用依赖。开发期的依赖是指在编译和连接阶段，构件之间的依赖关系。调用依赖是指一个构件调用或使用另外一个构件的服务。

例如，图 5-72 中，构件“窗口控制 (whnd.obj)”是构件“窗口控制 (whnd.cpp)”经过编译而建立的依赖关系。

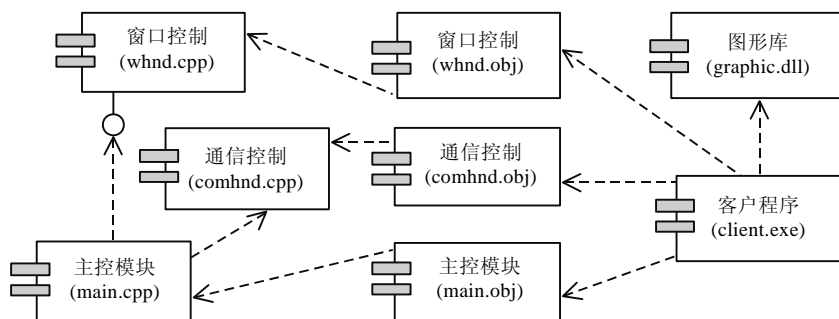


图 5-72 构件图实例

(2) 接口

接口也称为界面，是指构件对外提供的可见性操作和属性。通过接口，使一个构件可以访问另外一个构件中定义的操作。接口的图符用一个连接小圆圈的实线段来表示。在图 5-72 中，构件“窗口控制 (whnd.cpp)”为构件“主控模块 (main.cpp)”提供了一个接口，后者以依赖关系进行访问。使用接口具有更大的灵活性，有利于软件系统中构件的重用。

5.5.2 配置图

配置图 (Deployment Diagram) 描述了系统中硬件的物理配置情况，以及如何将软件部署到硬件上，描述了系统的体系结构，因此也称部署图。配置图的元素有结点和连接，连接描述了结点之间的通信类型。

1. 结点与连接

配置图中的结点代表计算机资源，通常是某种硬件，如服务器、客户机或其他硬件设备。结点包括在其上运行的软构件及对象。结点的图符是一个立方体。结点应标注名字，如图 5-73 所示。

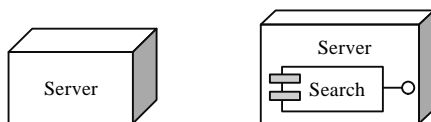


图 5-73 结点



配置图中各结点之间进行交互的通信路径称为连接。连接表示系统中的结点之间的联系，用结点之间的连线来表示连接，并且，在连接的连线上要标注通信类型。图 5-74 描述了一个保险系统的配置图，配置图中“客户 PC”结点和“保险服务器”结点是由通信路径按照 TCP/IP 协议连接的。

2. 构件与接口

软构件代表可执行的物理代码模块，如一个可执行程序。图 5-74 中，结点“保险服务器”包含了“保险系统”、“保险系统配置”和“保险数据库”三个构件。

在面向对象的方法中，并不是类和对象等元素的所有属性及操作对外都可见，它们对外提供的可见操作和属性称为接口，用一条连接小圆圈的线段表示。保险系统配置图中的“保险系统”构件，提供了一个称为“配置”的接口，图中还显示了构件之间的依赖关系，即“保险系统配置”构件通过接口依赖于“保险系统”构件。

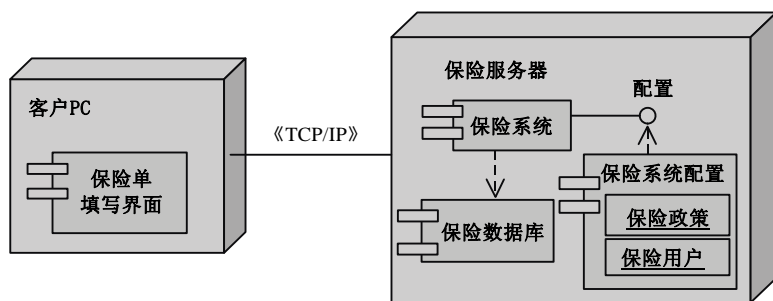


图 5-74 保险系统配置图

【例 5-12】现有一网上医院诊疗系统，病人在家中通过该系统就可以诊病。各科室的应用服务器与医院的数据库服务器相连接，从对象数据库中获取该科室诊病的相关知识和数据。这里以心血管病科室为例，建立医院诊疗系统的配置图。

首先确定系统由“数据库服务器”、“客户机”和“心血管病服务器”三个结点构成。“心血管病服务器”通过 TCP/IP 与结点“数据库服务器”和“客户机”连接。“数据库服务器”结点包括两个构件：“Object Database（对象数据库）”和“Health Care（心血管病领域）”。结点“客户机”中的构件“Heart Unit Client Façade（心血管病客户）”通过接口依赖于“心血管病服务器”结点中的构件“Heart Unit Server Application（心血管病应用程序）”，这样，病人通过网络获得心血管病服务器的服务，在网上看病。

图 5-75 是描述医院诊疗系统的配置图，是一个典型的三级 C/S 结构。请读者进一步分析各结点的构成和结点之间的关系。

并不是所有的系统都需要建立配置图，一个单机系统只需建立包图或构件图就行了。配置图主要用于在网络环境下运行的分布式系统或嵌入式系统的建模。

配置图可以显示计算机结点的拓扑结构和通信路径，结点上执行的软构件，软构件包含的逻辑单元等，特别是对于分布式系统，配置图可以清晰地描述系统中硬件设备的配置、通信及在各硬件设备上各种软构件和对象的配置。因此，配置图是描述任何基于计算机网络的

应用系统的物理配置或逻辑配置的有力工具。

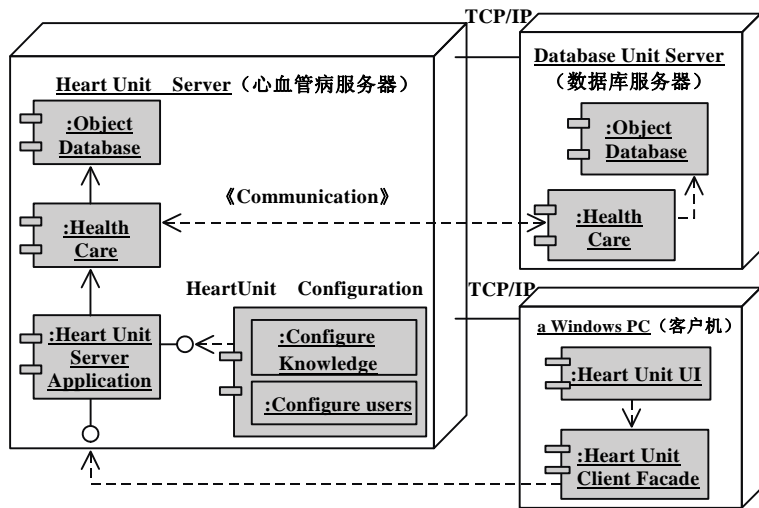


图 5-75 医院诊疗系统的配置图



5.6 统一过程及其应用

统一过程（Rational Unified Process, RUP）是由 Rational Software 公司首创的，因为它与当前流行的 Java、J2EE 技术和面向对象的设计思想（OOAD）紧密地结合在一起，所以在大型的信息技术项目中得到了广泛的应用。在本节中，我们试图对 RUP 进行一个初步的探讨，讨论它是如何贯穿在整个软件开发生命周期之中的。

RUP 又是一套软件工程方法的框架，RUP 与统一建模语言 UML 良好集成，支持多种 CASE 工具，并不断升级与维护，使之迅速得到业界广泛的认同。越来越多的组织以它作为软件开发模型框架。

RUP 吸收了多种开发模型的优点，具有很好的可操作性和实用性。这个过程的目的是在预定的进度和预算范围内，开发出满足最终用户需要的高质量软件。

5.6.1 UML与RUP

UML 需要软件过程。UML 能够用来为系统进行面向对象建模，但是并没有指定应用 UML 的过程，它是一种建模的语言，是独立于任何过程的。虽然 UML 的开发者在设计时是考虑了一些过程（见图 5-76），但对于有效地应用 UML，开发高质量的软件，是远远不够的。

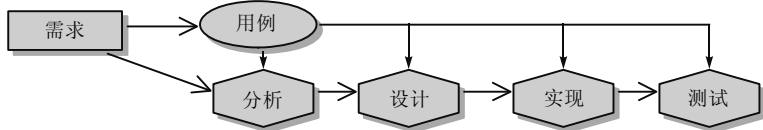


图 5-76 用例驱动的软件开发阶段



UML 建模过程的主要特征如下。

① 用例驱动的系统。用例包含了功能描述，它们将影响后面所有阶段及视图。

② 以体系结构为中心。在开发的早期建立基础的体系结构（原型）十分重要，通过进一步对原型进行精化，建立一个易于修改、易理解和允许复用的系统。其主要工作是在逻辑上将系统划分为若干个子系统（UML 包）。

③ 反复。UML 的建模过程要经过若干次的反复。

④ 渐增式。渐增式开发是指在多次反复迭代的过程中，每次增加一些功能（或用例）的开发，每次迭代都包含了分析、设计、实现和测试。

要想成功地应用 UML，应用一个好的软件过程是必要的。目前有很多的软件开发过程，如 RUP、OPEN Process、XP(Extreme Programming)、OOSP(Object_Oriented Software Process)等。其中，能够和 UML 最佳结合的是 RUP，不仅因为该过程的开发者也是 UML 的创立者，更因为 RUP 过程能够有效地测度工作进度，控制和改善工作效率。

5.6.2 RUP的特点

RUP 是最佳软件开发经验的总结，具有迭代式增量开发、使用实例驱动和以软件体系结构为核心这三个鲜明特点。这些特点是对 UML 的发展和无缝集成。

RUP 包含了软件开发中的六大经验：迭代式开发，管理需求，使用基于组件的软件体系结构，可视化建模，验证软件质量，控制软件变更。

(1) 迭代式开发 (Develop Iteratively Develop Iteratively)

在软件开发的早期就想完全、准确地捕获用户的需求几乎是不可能的。实际上，我们经常遇到的问题是，需求在整个软件开发过程中经常会变化。迭代式开发允许在每次迭代过程中需求发生变化，通过不断细化来加深对问题的理解。因此迭代式开发不仅可以降低项目的风险，而且每次迭代过程可以产生新版本。

(2) 管理需求 (Manage Requirements)

确定系统的需求是一个连续的过程。开发人员在开发系统之前不可能完全详细地说明一个系统的真正需求。RUP 描述了如何提取、组织系统的功能和约束条件并将其文档化，用例和脚本的使用已经被证明是捕获功能需求的有效方法。

(3) 应用基于组件的体系结构 (Use Component Architectures)

组件使重用成为可能，系统可以由组件构成。基于独立的、可替换的、模块化组件的体系结构有助于控制系统的复杂性，提高重用率。RUP 描述了如何设计一个有弹性的、能适应变化的、易于理解的、有助于重用的软件体系结构。

(4) 可视化建模 (Use Component Architectures)

RUP 和 UML 联系在一起，对软件系统进行可视化建模，帮助人们提供管理和控制软件复杂性的能力。

(5) 验证软件质量 (Continuously Verify Quality)

在 RUP 中，软件质量评估不再是软件开发完成后才进行的活动，而是贯穿于软件开发过程中，这样可以及早发现软件中的缺陷并予以纠正。

(6) 控制软件变更（Manage Change）

RUP 描述了如何进行控制、跟踪、监控、修改，以确保成功的迭代开发。RUP 通过软件开发过程中的制品，将软件的变更控制在最小范围内，并以此为每个开发人员建立安全的工作空间。

总之，在 RUP 中，以用例捕获需求方法的优势是显而易见的。首先，它描述了用户是如何与系统交互的，这种描述更易于被用户所理解，是开发人员和用户之间针对系统需求进行沟通、迅速达成共识的有效手段。其次，由于它是以时间顺序描述交互过程的，因此系统分析员和用户都可以轻易地识别用例中存在的缺陷。再次，它能使团队成员在设计、实现、测试和最后编写用户手册的过程中紧紧地以用户需求为中心，促使开发人员始终站在用户的角度考虑问题，容易验证设计和实现满足用户的需求。此外，用例还简化了记录功能需求的工作，提高了开发工作的效率。

5.6.3 RUP的二维开发模型

传统的软件开发模型，如瀑布式开发模型，是一维的模型，开发工作被划分为多个连续的阶段，在一个时间段内，只能做某一个阶段的工作，如分析、设计或者实现。

如图 5-77 所示，在 RUP 中，软件开发生命周期根据时间和 RUP 的核心工作流程划分为二维空间：横轴描述 RUP 开发过程的动态结构，纵轴描述 RUP 的静态组成部分。

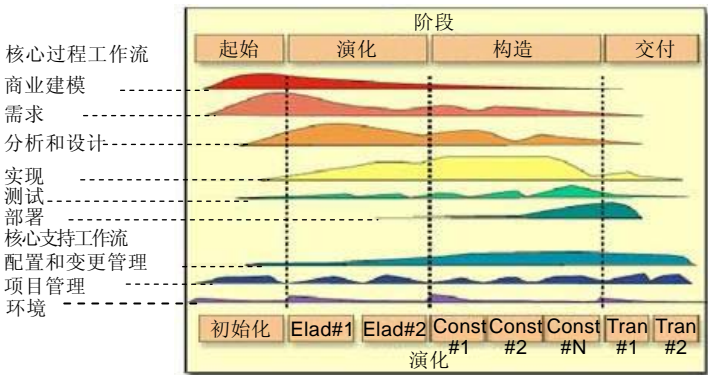


图 5-77 RUP 二维开发模型

1. 横轴

横轴为时间轴。时间轴从组织管理的角度描述整个软件开发生命周期，是 RUP 的动态组成部分。RUP 把软件开发周期划分为起始（Inception）、演化（Elaboration）、构造（Construction）和交付（Transition）4 个连续的阶段。

(1) 起始阶段

该阶段的目标是为系统建立商业案例并确定项目的边界，为此，必须识别所有与系统交互的外部实体，在较高层次上定义交互的特性。本阶段具有非常重要的意义，在这个阶段中所关注的是整个项目进行中的业务和需求方面的主要风险。



(2) 演化阶段

其目标是分析问题领域, 在理解整个系统的基础上, 建立系统的体系结构, 包括其范围、主要功能和诸如性能等非功能需求, 并编制项目计划。

(3) 构造阶段

所有剩余的构件和应用程序功能被开发并集成为产品, 所有的功能被详细测试。从某种意义上说, 构建阶段是一个制造过程, 其重点放在管理资源及控制运作上, 以优化成本、进度和质量。该阶段的产品版本被称为 **Deta** 版。

(4) 交付阶段

交付阶段的重点是确保软件对最终用户是可用的, 常常要进行几次迭代, 包括为发布做准备的产品测试, 基于用户反馈的少量的调整等。

2. 纵轴

纵轴表示核心 workflow。workflow 描述了一个有意义的连续的行为序列, 每个 workflow 产生一些有价值的产品, 并显示了角色之间的关系。核心 workflow 从技术角度描述 **RUP** 的静态组成部分。**RUP** 中的 9 个核心 workflow (**Core Workflows**) 如下。

(1) 商业建模 (**Business Modeling**)

商业建模 workflow 理解待开发系统的组织结构及其商业运作, 建立商业用例模型和商业对象模型, 定义组织的过程、角色和责任, 评估待开发系统对结构的影响, 确保所有参与人员对待开发系统有共同的认识。

(2) 需求 (**Requirements**)

需求 workflow 的目标是描述系统应该做什么, 定义系统功能及用户界面, 为项目预算及计划提供基础, 并使开发人员和用户就这一描述达成共识。

(3) 分析和设计 (**Analysis & Design**)

分析和设计 workflow 要将需求分析的结果转化为实现规格。分析设计的结果是一个设计模型和一个可选的分析模型。设计模型是源代码的抽象, 由设计类和一些描述组成。设计类应具有良好接口的设计包 (**Package**) 和设计子系统 (**Subsystem**), 而描述则体现了类的对象如何协同工作实现用例的功能。

设计活动以体系结构设计为中心, 体系结构由若干结构视图来表达, 结构视图是整个设计的抽象和简化 (省略了一些细节), 使重要的特点体现得更加清晰。体系结构不仅仅是良好设计模型的承载媒介, 而且在系统的开发中能提高被创建模型的质量。

(4) 实现 (**Implementation**)

实现 workflow 的目的是定义代码的组织结构、实现代码、单元测试、系统集成, 以组件的形式 (源文件、二进制文件、可执行文件) 实现类和对象, 使其成为可执行的系统。

(5) 测试 (**Test**)

测试 workflow 要验证各自子系统的交互与集成, 确保所有的需求被正确实现, 并在系统发布前发现错误和改正错误。

RUP 提出了迭代的方法, 意味着要在整个项目中进行测试, 从而尽可能早地发现缺陷, 从根本上降低了修改缺陷的成本。测试类似于三维模型, 分别从可靠性、功能性和系统性能来进行。

（6）部署（Deployment）

部署 workflows 描述确保软件产品对最终用户具有可用性的相关活动，包括：打包、分发、安装软件，升级旧系统；培训用户及销售人员进行，并提供技术支持；制定并实施 Delta 测试；移植现有的软件和数据以及正式验收。

（7）配置和变更管理（Configuration & Change Management）

配置和变更管理工作流跟踪并维护系统开发过程中产生的所有制品的完整性和一致性。同时也阐述了对产品的修改原因、时间、人员，并保持审计记录。

（8）项目管理（Project Management）

软件项目管理为软件开发项目提供计划、人员配备、执行、执行和监控项目提供实用的准则，为风险管理提供框架。

（9）环境（Environment）

环境工作流的目的是向软件开发组织提供过程管理和工具的支持。环境工作流集中于配置项目过程中所需要的活动，同样也支持开发项目规范的活动，提供步骤的指导手册并介绍如何在组织中实现过程。

这 9 个核心工作流分为两种组织工作流的方式：（1）～（6）为核心过程工作流（Core Process Workflows）方式，（7）～（9）为核心支持工作流（Core Supporting Workflows）。

从图 5-77 中用阴影表示的工作流可以看出，不同的工作流在不同时间段内的工作量不同。值得注意的是，几乎所有的工作流，在所有的时间段内，均有工作量，只是工作程度不同而已。这与（Waterfall Process 瀑布式开发）模型有着明显的不同。9 个核心工作流在项目迭代开发过程中轮流被使用，在每一次迭代中以不同的重点和强度重复。

需要说明的是，RUP 的 9 个核心工作流并不总是需要的，可以取舍。通过对 RUP 进行裁剪可以得到很多不同的开发过程。这些软件开发过程可以看做 RUP 的具体实例，根据本项目具体情况确定需要哪些工作流。

5.6.4 RUP 的迭代开发模式

在 RUP 的二维开发模型中，每个阶段都由一个或多个连续的迭代组成，每一个迭代都是一个完整的开发过程，产生一个可执行的产品版本，是最终产品的一个子集，如图 5-78 所示，它增量式地从一个迭代过程到另一个迭代过程直到成为最终的系统。

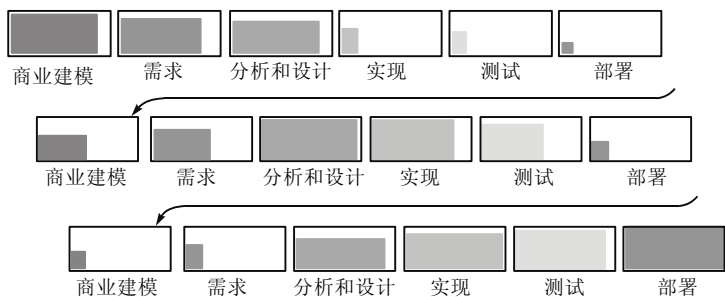


图 5-78 RUP 迭代过程



如图 5-79 所示，在每个阶段结束前都应有一个里程碑（MileStone），用来评估该阶段的工作，只有当阶段目标达到时才允许项目进入下一阶段，产生一个阶段里程碑。若未能通过评估，则决策者应该做出决定，是应取消该项目，还是继续做该阶段的工作。

所以这是一种更灵活、风险更小的方法，多次通过不同的开发工作流，这样可以更好地理解需求，构造一个健壮的体系结构，并最终交付一系列逐步完成的版本。这称为一个迭代生命周期。

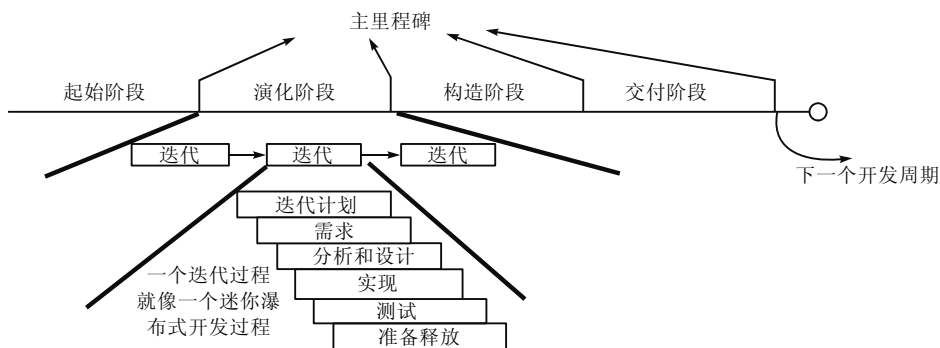


图 5-79 RUP 分阶段评估

与核心工作流不同的是，RUP 并没有，也无法给出迭代工作流的具体实现步骤，它需要项目经理根据当前迭代所处的阶段以及上次迭代的结果，适当地对核心工作流中的行为进行裁剪以实现一个具体的可操作的迭代工作流。

由于 RUP 的开发过程是以软件的体系结构为中心，以用例驱动的，因此 RUP 的迭代开发过程是可控制的。在项目计划中已制定了项目迭代的次数、每个迭代的延续时间以及目标。在每一次迭代的起始阶段都制定详细的迭代计划及具体的迭代工作流。每次迭代过程都生成该次迭代的版本，作为下次迭代的基础。在迭代结束前，都应执行测试工作评估该迭代过程，为下一次迭代做准备。迭代不是重复，而是针对不同用例的细化和实现。

习题 5

- 5.1 简述扩展、包含和细化三种 UML 依赖关系的异同。
- 5.2 软件开发为什么要使用 UML 建模？它有何特点？
- 5.3 简述 UML 实际建模过程。
- 5.4 在 UML 中的状态图、协作图、活动图、序列图在系统分析中各起到了什么作用？
- 5.5 顺序图与合作图都是交互图，它们有何不同？所描述的主要系统特征是什么？
- 5.6 状态图与活动图有何相同与不同之处？在建立系统模型时，应该如何使用这两类模型？
- 5.7 什么是抽象类？在建模时使用抽象类有什么好处？
- 5.8 在分析和设计阶段都需要建立类图，试说明分析类图与设计类图的主要区别是什么？
- 5.9 问题描述为：储户用存折取款，首先填写取款单，根据“账卡”中的信息检验取

款单与存折，若有问题，则将问题反馈给储户，否则，登录“储户存款数据库”，修改相应数据，并更新“账卡”，同时发出付款通知，出纳向储户付款。

- (1) 建立系统的用例模型；
- (2) 建立角色和用例的描述模板。

5.10 一个小型图书资料管理系统的主要功能为：图书资料的借出、归还、查询和管理，该系统包括图书管理员和普通读者，普通读者要使用系统必须先注册。

图书管理员负责添加、更新和修改、删除图书资料，登记和查询图书的借阅、归还情况。读者可以按照作者或主题检索图书资料，还可以预订图书资料，即当新购买或有读者归还时，系统立即通知读者来借阅。

- (1) 确定系统的类，并定义其属性和操作；
- (2) 画出系统的分析类图。

第6章 软件测试



6.1 软件测试概述



6.1.1 软件测试的基本概念

在软件系统的分析、设计、编码等开发过程中，尽管开发人员采取了许多保证软件产品质量的手段和措施，但是错误和缺陷仍然是不可避免的。例如，对用户需求理解不正确、不全面，实现过程中的编码错误等。这些错误和缺陷，轻者导致软件产品无法完全满足用户的需要，重者导致整个软件系统无法正常运行，造成巨大的损失和浪费。因此，为了确保软件产品的质量，在软件开发的一系列过程中及时发现并纠正错误是十分重要的。软件测试是在软件开发过程中保证软件质量、提高软件可靠性的最主要的手段之一，它是在软件产品在交付用户使用之前，对分析、设计、编码等开发工作的最后检查和复审。

无论如何强调软件测试对于确保软件系统的质量的重要性都不过分。根据相关开发组织的大量统计资料，在整个软件系统的开发过程中，软件测试占了其中 40%~50% 的工作量。特别是在一些特殊或重要的软件系统中，例如武器火控系统、核反应控制系统、航空/航天飞行系统等，软件测试环节的工作量和成本往往是其他所有开发活动总工作量的 3~5 倍。

为了确切地描述软件测试的含义，根据 IEEE（1983）标准，先定义以下几个重要概念。

- ◎ 测试：选择适当的测试用例执行被测试程序的过程，它的目的在于发现程序错误。
- ◎ 调试：诊断程序的错误性质、出错位置并加以改正的过程，通常由编码人员承担。
- ◎ 失败：当一个程序不能运行时称为失败。失败是系统执行中出现的情况，失败源于代码缺陷。
- ◎ 错误：由于程序中的缺陷所产生的不正确的结果称为错误。

程序中的人为缺陷可导致系统失败（程序不能运行），也可能出现错误结果（程序可运行）。

正如 E.W.Dijkstra 所指出的：“测试只能证明程序有错（有缺陷），不能保证程序无错”。因此，能够发现程序缺陷的测试是成功的测试。当然，最理想的是进行程序正确性的完全证明，遗憾的是，除了极小的程序外，至今还没有实用的技术能证明任一程序的正确性。为使程序有效运行，测试与调试是唯一手段。

测试的根本目的就是发现尽可能多的缺陷。这里的缺陷是一种泛称，它可以指功能的错误，也可以指性能低下，易用性差等。因此，测试是一种“破坏性”行为（不破不立）。

对软件测试，Glen Myers 提出了下述观点：

- ① 测试是一个程序的执行过程，其目的是发现错误。
- ② 一个好的测试用例很可能发现至今尚未发现的错误。
- ③ 一个成功的测试用例可以发现至今尚未发现的错误。

主观上，由于开发人员思维的局限性，客观上，由于目前开发的软件系统都具有相当的复杂性，决定了在开发过程中出现软件错误是不可避免的。若能及早排除开发中的错误，就可以减少给后期工作带来的麻烦，也就避免了付出高昂的代价，从而大大地提高系统开发过程的效率。因此，软件测试在整个软件开发生命周期的各个环节中都是不可缺少的。

1994 年在“软件工程师参考手册”中，J.McDermid 提出了“生存周期软件开发 V 模型”，进一步说明了测试的重要性。此后，不少学者在此基础上做了很多介绍。现结合 ISO/IEC 12207:1995 “软件生存周期过程”做了一些调整，如图 6-1 所示。需要说明的是，生存周期软件开发 V 模型并不针对某种开发模型或某种开发方法，它是按照软件生存周期的不同阶段划分的，因此不能认为它只适合于瀑布模型。

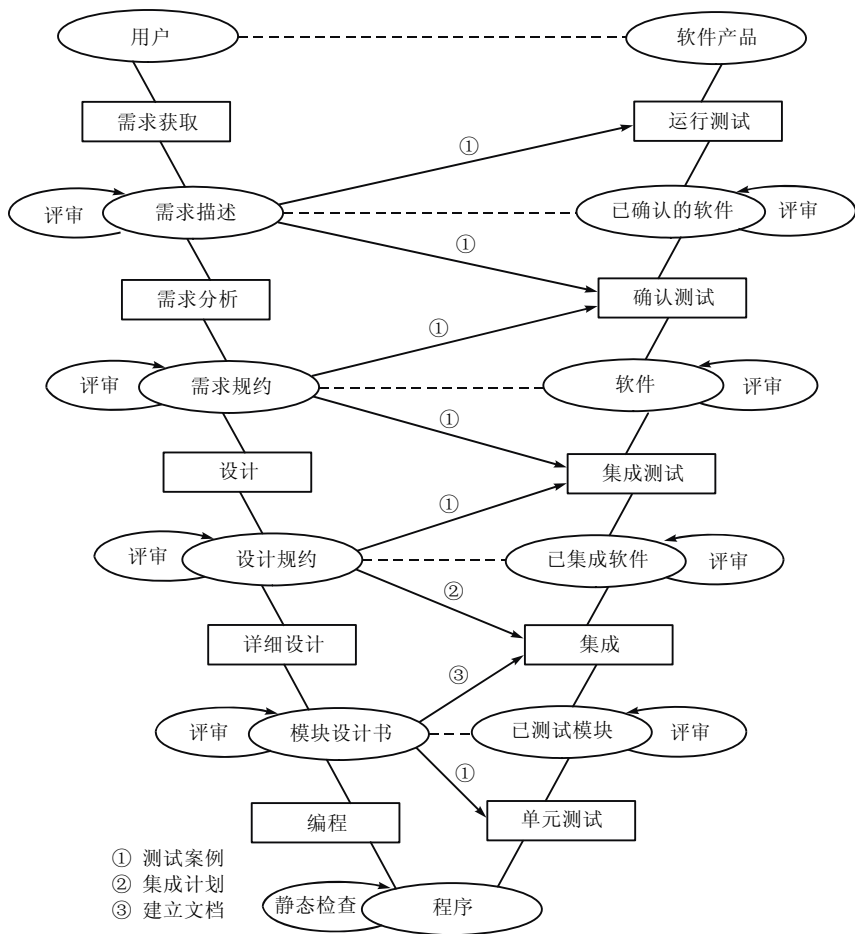


图 6-1 生存周期软件开发 V 模型

总体来说，软件测试的目标在于以最小的工作量和成本尽可能多地发现软件系统中潜在的各种错误和缺陷，以确保软件系统的正确性和可靠性。在软件测试工作中，建立正确的测



试目标具有重要的价值和意义。如果软件测试的目标是证明软件的正确性,那么测试人员就会选用那些使软件出错可能性较小的数据作为测试用例;如果软件测试的目标是证明软件有错误,那么测试人员就会选用那些容易使软件发生错误的数据作为测试用例,以尽可能多地发现软件系统中潜在的错误和缺陷。因此,从心理学的角度来看,一个测试人员的工作是要证明和发现软件有错,只有选用那些易使程序出错的测试用例,才能够有效地发现程序中的错误和缺陷,达到软件测试的目的。

6.1.2 软件测试的特点和基本原则

1. 软件测试的特点

(1) 软件测试的开销大

按照 Boehm 的统计,软件测试的开销大约占总成本的 30%~50%。例如,阿波罗登月计划,80%的经费用于软件测试。

(2) 不能进行“穷举”测试

只有将所有可能的情况都测试到,才有可能检查出所有的错误,但这是不可能的。

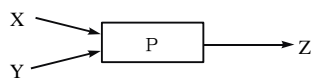


图 6-2 程序 P

【例 6-1】程序 P 有两个整型输入量 X、Y,输出量为 Z (见图 6-2),在 32 位机上运行。所有的测试数据组 (X_i, Y_i) 的数目为: $2^{32} \times 2^{32} = 2^{64}$ 。假设 1ms 执行 1 次,要进行完全测试,共需 5 亿年。

2. 软件测试的基本原则

测试是一项非常复杂的、创造性的且需要高度智慧的挑战性的工作。测试一个大型程序所要求的创造力,事实上可能要超过设计那个程序所要求的创造力。软件测试中一些直观上看是显而易见的、至关重要的原则,总是被人们忽视。

(1) 应尽早地、不断地进行软件测试

相关的研究数据表明,软件系统的错误和缺陷具有明显的放大效应。在需求阶段遗留的一个错误,到了设计阶段可能导致出现 n 个错误,而到了编码实现阶段则可能导致更多的错误。因此,在许多成熟的软件开发模型中,软件测试不再是整个软件开发完成以后才进行的活动。软件测试不同于程序测试,在软件系统的整个开发周期的各个阶段中,都应该始终贯穿软件测试活动,只有这样,才能尽早地发现潜在的错误和缺陷,降低软件测试的成本,提高软件测试的质量。

(2) 开发人员应尽量避免进行软件测试

开发和测试从来都是不同的活动。开发是创造或建立新事物的行为活动;而测试的唯一目的是证明开发的软件产品运行不正常,存在若干潜在的错误和缺陷。因此,在这两个活动之间存在着本质的对立和矛盾。一个开发人员不可能同时将这两个截然对立的角色扮演好。

具体来说,开发者在测试自己的程序时存在以下弊病。

① 开发者对自己的程序印象深刻,并总以为是正确的。倘若在设计时就存在理解错误,或因不良的编程习惯而留下隐患,那么他本人是很难发现这类错误的。

② 开发者对程序的功能、接口十分熟悉，他自己几乎不可能因为使用不当而引发错误，这与大众用户的情况不同，所以自己测试程序不具备典型性。

③ 程序设计犹如艺术设计，开发者总是喜欢欣赏程序的成功之处，而不愿看到失败之处。让开发者去做“蓄意破坏”的测试，就像扼杀自己的孩子一样难以接受。即便开发者非常诚实，但“珍爱程序”的心理让他在测试时不知不觉地带入了虚假成分。

下面我们再来看看 Microsoft 公司关于测试的经验教训。

在 20 世纪 80 年代初期，Microsoft 公司的许多软件产品出现了 Bug。例如，在 1981 年与 IBM PC 一起推出的 BASIC 软件，用户在用“.1”（或者其他数字）除以 10 时，就会出错。在 FORTRAN 软件中也存在破坏数据的 Bug。由此激起了许多采用 Microsoft 操作系统的 PC 厂商的极大不满，而且很多个人用户也纷纷投诉。

Microsoft 公司的经理们发现，必须引进更好的内部测试与质量控制方法。但是这遭到很多程序设计师甚至一些高级经理的坚决反对，他们固执地认为在高校学生、秘书或者外界合作人士的协助下，开发人员可以自己测试产品。在 1984 年推出 Mac 机的 Multiplan（电子表格软件）之前，Microsoft 曾特地请 Authur Anderson 咨询公司进行测试。但是外界公司一般没有能力执行全面的软件测试。结果是，一种相当厉害的破坏数据的 Bug 迫使 Microsoft 公司为它的 2 万多名用户免费提供更新版本，代价是每个版本 10 美元，一共花了 20 万美元，可谓损失惨重。

痛定思痛后，Microsoft 公司的经理们得出一个结论，如果不成立独立的测试部门，软件产品就不可能达到更高的质量标准。IBM 公司和其他有着成功软件开发历史的公司便是效法的榜样。但 Microsoft 公司并不照搬 IBM 公司的经验，而是有选择地采用了一些看起来比较先进的方法，如独立的测试小组，自动测试以及为关键性的构件进行代码复查等。Microsoft 公司开发部门的主管戴夫·穆尔回忆说：“我们清楚不能再让开发部门自己测试了。我们需要一个单独的小组来设计测试，运行测试，并把测试信息反馈给开发部门。这是一个伟大的转折点。”

从以上的例子可以看出，由独立的测试机构来完成软件系统测试工作具有许多显著的优点。独立测试是指软件测试工作由在管理上和经济上独立于开发机构的组织来承担。独立测试可以避免软件开发机构测试自己开发的软件。由于受到时间、成本和质量的制约与影响，软件产品在开发过程中的质量容易遭到忽视，如果测试组织与开发组织来自相同的机构，那么测试工作就会面临与开发组织同一起来源的管理方面的压力，使测试工作受到影响和干扰。

（3）注重测试用例的设计和选择

测试用例由输入数据和预期的输出数据两部分组成，是整个软件产品各阶段活动的主体，因此测试用例设计至关重要。测试用例的质量可以用以下 4 个特性来描述。

- ◎ 有效性：能否发现软件缺陷或至少可能发现软件缺陷。
- ◎ 可仿效性：可仿效的测试用例可以测试多项内容，从而减少测试用例数量。
- ◎ 经济性：测试用例的执行分析和排错是否经济。
- ◎ 修改性：每次软件修改后对测试用例的维护成本。

这 4 个特性之间会有影响，例如，高仿效性有可能导致经济性和修改性较低。因此，在通常情况下，应对上述 4 个特性进行一定的权衡折中。



在设计测试用例时，需要格外注意以下两个问题。

① 测试用例应该不仅包含合理的输入条件，更应该包含不合理的输入条件。在软件系统的实际运行过程中，经常会发生这样一种情况，当以某种特殊的甚至是不合理的输入数据使用软件时，常常会发生许多意想不到的错误。因此，使用预期不合理的输入数据进行软件测试，经常比使用合理数据会有更大的收获。

② 测试用例应由测试输入数据和与之对应的预期输出结果这两部分组成。这些期望的输出结果应该是根据系统的需求来进行定义的，因此测试人员在将系统的实际输出与测试用例中的预期输出进行对比以后，就可以完成对软件系统正确性、可靠性的测试，发现其中是否存在相应的错误和缺陷。

（4）充分注意测试中的群集现象

软件系统中的错误和缺陷通常不是均匀地分布在整个软件系统的各个部分的，而是成群集中出现，经常会在一个模块或一段代码中存在大量的错误和缺陷。例如，在 IBM370 的某个操作系统中，发现的错误和缺陷有 47% 集中在 4% 的代码中，这一现象的出现告诉测试人员，为了提高软件的测试效率，要集中处理那些容易出现错误的模块或程序段。具体地说，群集现象是指，在测试过程中，发现错误比较集中的程序段，往往可能残留的错误数较多。因此必须注意这种群集现象，对错误群集的程序段进行重点测试，以提高测试的效率和质量。

（5）避免测试的随意性，严格执行测试计划

测试人员或机构在进行软件测试工作之前，应该制定详细、完善的测试计划。测试计划是指对测试范围、测试方式、测试成本、测试工作量、测试时间等内容做出一个预先的规划。测试计划的内容应该包含进行测试的项目，每位测试人员应该负责的测试工作，以及测试的风险和进度。测试计划的书写应该明确清晰、无二义性，而在测试工作进行过程中，测试机构和测试人员应该严格按照该计划执行，避免测试工作的随意性，只有这样，才能保证对软件产品进行系统、科学的测试。

（6）全面检查每一个测试结果

在使用测试用例对软件产品进行测试时，对每一个测试结果应该做全面、细致的检查。因为许多错误的迹象和线索会在输出结果中反映或表现出来。不对测试结果进行全面、细致的检查，这些有用的测试信息将会被遗漏掉，严重影响软件测试的质量和效率。

（7）妥善保存测试过程中的一切文档，为软件维护提供方便

测试计划、测试用例、测试结果、出错统计等都是软件测试过程中的重要文档。和软件开发过程中其他阶段的文档一样，这些文档的完整、妥善保存都是十分重要的，它可以为后期的软件维护提供许多便利，例如，测试的重现往往要依靠测试文档的相关内容。

另外，Davis 提出了一组测试原则，在设计有效地测试用例时必须理解。

① 所有的测试都应根据用户的需求来进行。

② 应该在测试工作真正开始前的较长时间内就进行测试计划（测试规划）。一般而言，测试计划可以在需求分析完成后开始，详细的测试用例定义可以在设计模型被确定后立即开始，因此，所有测试可以在任何代码被编写前进行计划 and 设计。

③ 将 Pareto 原则应用于软件测试。Pareto 原则是，测试发现错误的 80% 集中在 20% 的程序模块中。

- ④ 测试应从“小规模”开始，逐步转向“大规模”，即从模块测试开始，再进行系统测试。
- ⑤ 穷举测试是不可能的，因此，在测试中不可能覆盖路径的每一个组合，然而，充分覆盖程序逻辑，确保覆盖程序设计中使用的所有条件是有可能的。
- ⑥ 为达到最佳的测试效果，提倡由第三方来进行测试。

6.1.3 软件测试过程

1. 软件测试的基本步骤

在软件的测试活动中，存在着一种误解，即将软件测试等同于程序测试，认为软件测试的对象仅限于源程序代码。实际上，软件测试的对象应该包括需求分析、概要设计、详细设计、编码实现各个阶段所获得的开发成果，程序测试仅仅是软件测试的一个组成部分，软件测试应该贯穿于整个软件开发的全过程。

根据相关的统计资料发现，在查找出的软件错误中，大约有 64% 的错误都属于软件的需求分析和设计阶段。这一统计结果表明，对于软件系统而言，许多都是因为前一阶段的错误没有被及时发现，而导致后续错误的发生。

因此，为了避免软件开发过程前一阶段的错误影响后续的开发活动，开发人员必须实施分阶段、分步骤的测试，以确保软件开发过程各个阶段产品的质量。软件测试过程可按以下步骤进行：单元测试、集成测试、确认测试和系统测试，最后进行验收测试，如图 6-3 所示。

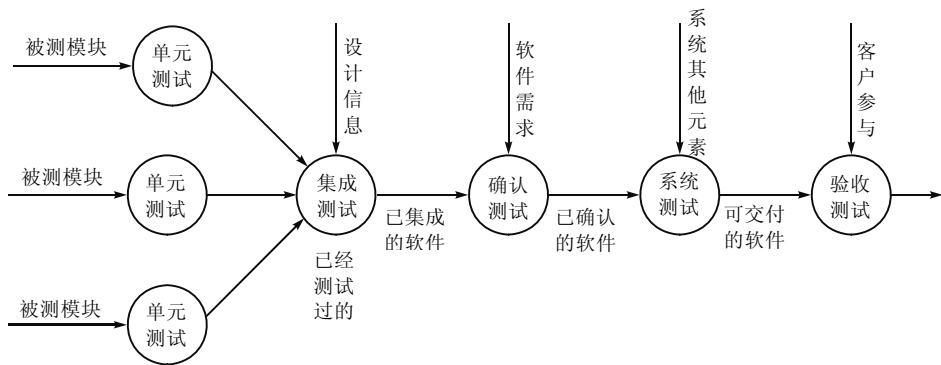


图 6-3 软件测试的过程

- ① 单元测试：分别完成每个单元的测试任务，以确保每个模块能正常工作。单元测试大量地采用了白盒测试方法，尽可能发现模块内部的程序差错。
- ② 集成测试：把已测试过的模块组装起来，进行集成测试。其目的在于检验与软件设计相关的程序结构问题。这时较多地采用黑盒测试方法来设计测试用例。
- ③ 确认测试：完成集成测试以后，要对开发工作初期制定的确认准则进行检验。确认测试是检验所开发的软件能否满足所有功能和性能需求的最后手段，通常均采用黑盒测试方法。
- ④ 系统测试：完成确认测试以后，给出的应该是合格的软件产品，但为检验它能否与系统的其他部分（如硬件、数据库及操作人员）协调工作，需要进行系统测试。严格地说，系统测试已超出了软件工程的范围。



⑤ 验收测试：检验软件产品质量的最后一道工序是验收测试。与前面讨论的各种测试活动的不同之处主要在于，它突出了客户的作用，同时软件开发人员也应进行一定程度的参与。

软件测试工作从概要设计阶段就开始了，如图 6-4 所示，整个测试分为两个大的阶段：预测试阶段和测试阶段。本章只考虑测试阶段的工作。在进行具体测试时，有些步骤可以合并，例如，功能测试与系统测试。

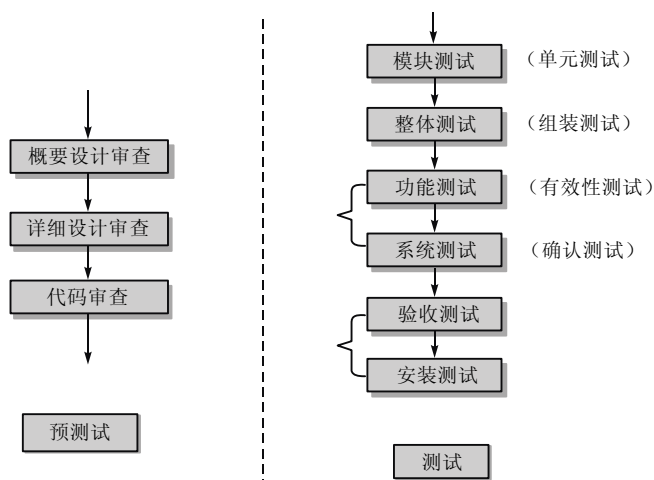


图 6-4 软件测试步骤

2. 软件测试的文档

软件测试工作是一个非常复杂而艰巨的过程，同时涉及了需求分析、设计、编码等许多软件开发的其他环节。同时，软件测试工作对于保证软件的正确性、可靠性、健壮性具有十分重要的意义。因此，必须将软件测试的要求、过程、结果以正式文档的形式加以记录。测试文档的撰写是软件测试工作规范化的一个重要组成部分。在软件系统的测试工作中，主要的测试文档如下。

(1) 测试计划

测试计划是软件测试工作的指导性文档，它规定了测试活动的范围、测试方法、测试的进度与资源、测试的项目与特性。在测试计划中，应明确需要完成的测试任务、每个任务的负责人，以及与测试活动相关的风险。

具体地说，测试计划一般包括：测试目标、测试范围、测试方法、测试资源、测试环境和工具、测试体系结构、测试进度。

(2) 测试规范

测试规范规定了测试工作的一些总体原则并描述了测试工作的一些基本情况，例如，测试用例的运行环境、测试用例的生成步骤与执行步骤、软件系统的调试与验证。

(3) 测试用例

测试工作通常需要设计若干测试用例，每个测试用例包括一组测试数据和一组预期的运行结果。因此，一个典型的测试用例可以被描述为：

测试用例={测试数据+期望的运行结果}

相应地，测试结果可以被描述为：

测试结果={测试数据+期望的运行结果+实际的运行结果}

(4) 缺陷报告

缺陷报告主要用于记录在测试过程中发现的软件系统存在的错误与缺陷，具体包括：缺陷编号、缺陷的严重程度和优先级、缺陷的状态、缺陷发生的位置、缺陷的报告步骤、期待的修改结果以及附件等内容。

在缺陷报告中，需要特别说明的是缺陷的严重程度和优先级。需要注意的是，这是两个不同的概念。其中，缺陷的严重程度是指缺陷的恶劣程度，反映其对整个软件系统和用户的危害程度；缺陷的优先级是指纠正这一缺陷在时间上的紧迫程度。

3. 软件测试的人员

测试工作是一个艰巨而复杂的过程，是保障软件质量的重要屏障，因此它对进行测试的人员提出了严格的要求。缺乏一个合格、积极的测试团队，测试任务是无法圆满完成的，这必然会严重地影响整个软件产品的质量。

在许多软件开发企业中，特别是一些小型的、不成熟的软件企业中，对软件测试工作的重视程度不够，有一种非常普遍的习惯，就是让那些熟练的开发人员完成软件系统的分析、设计、实现等工作，而让那些开发经验最少的新手去承担被认为相对次要的软件测试工作。这种做法是非常不合理且不科学的。对一个软件系统进行高效率、高质量的测试所需要的技能与经验其实并不比开发一个新软件需要得少。因此，在一些比较成熟的软件企业中，都将软件测试看做是一项专业的技术工作，有意识地在开发团队中培训专门的软件测试人员，并在开发过程中及时地投入工作，以便完成高质量的软件测试。

6.1.4 静态分析与动态测试

1. 静态分析

静态方法的主要特征是，在用计算机测试源程序时，计算机并不真正运行被测试的程序。这说明静态方法一方面要利用计算机作为被测程序进行特性分析的工具，它与人工测试有着根本的区别；另一方面，它并不真正运行被测程序，只进行特性分析，这与动态测试不同。因此，静态方法常称为“静态分析”。静态分析是对被测程序进行特性分析的一些方法的总称。

静态分析并不等同于编译系统，编译系统虽也能发现某些程序错误，但这些错误远非软件中存在的大部分错误，静态分析的查错和分析功能是编译程序所不能代替的。目前，已经开发出一些静态分析系统作为软件测试的工具，静态分析被当做一种自动化的代码校验方法。不同的方法有各自不同的目标和步骤，侧重点也不一样。常用的静态测试方法如下。

(1) 桌前检查 (Desk Checking)

由程序员检查自己的程序，对源代码进行分析、检验。

作为一种传统的检查方法，桌前检查常常在程序通过编译以后，进行单元测试之前进行，对源程序中的代码进行分析、检验，并补充相应的文档，以发现程序中潜在的错误和缺陷。桌前检查的具体检查项目包括：检查变量的交叉引用是否正确，检查标号的交叉引用是否正



确，检查子程序或函数的调用是否正确等。

(2) 代码会审 (Code Reading Review)

由程序员和测试员组成评审小组，按照“常见的错误清单”，进行会议讨论检查。

代码会审一般分为两个步骤。第一，有评审小组的负责人提前将软件系统的设计规格说明书、控制流程图等相关文档分发给与会审的程序员和测试员，这些文档将作为会审的依据，评审小组的成员在参与会审之前需要熟悉这些文档资料；第二，召开代码会审会议，在会上，开发人员讲解软件系统的分析、设计与实现，而评审人员可以提出质疑，展开相应的讨论，通过这样一种评审人员与开发人员的相互讨论与交流，软件系统中需要隐藏的 errors 和缺陷可能会暴露出来，以此实现对软件产品的测试，确保软件产品的质量。

(3) 步行检查 (Walkthroughs)

与代码会审类似，也要进行代码评审，但评审过程主要采取人工执行程序的方式，故也称为“走查”。

步行检查是最常用的静态分析方法，进行步行检查时，还常使用以下分析方法。

◎ 调用图：从语义的角度考察程序的控制路线；

◎ 数据流分析图：检查分析变量的定义和引用情况。

2. 动态测试

动态测试的主要特征是计算机必须真正运行被测试的程序，通过输入测试用例，对其运行情况（输入/输出的对应关系）进行分析。

动态测试方法与静态分析方法的区别是：动态测试方法需要通过选择适当的测试用例，上机执行程序进行测试。常用的方法如下。

① 白盒测试 (White-box Testing)：又称结构测试、逻辑驱动测试或基于程序的测试。它依赖于对程序细节的严密检验，针对特定条件或循环集设计测试用例，对软件的逻辑路径进行测试。因此采用白盒测试技术时，必须有设计规约及程序清单。设计的宗旨是，测试用例尽可能提高程序内部逻辑的覆盖程度，最彻底的白盒测试要能够覆盖程序中的每一条路径。但是如果程序中含有循环，那么路径的数目极多，要执行每一条路径变得极不现实。软件的白盒测试用来分析程序的内部结构。

② 黑盒测试 (Black-box Testing)：又称功能测试、数据驱动测试或基于规格说明的测试，是一种从用户观点出发的测试。用这种方法进行测试时，把被测程序当做一个黑盒，在不考虑程序内部结构和内部特性，测试者只知道该程序输入和输出之间的关系或程序的功能的情况下，依靠能够反映着这一关系和程序功能需求规格的说明书，来确定测试用例盒推断测试结果的正确性。软件的黑盒测试被用来证实软件功能的正确性和可操作性。

无论白盒测试还是黑盒测试，关键都是如何选择高效的测试用例。所谓高效的测试用例，是指一个用例能够覆盖尽可能多的测试情况，从而提高测试效率。白盒测试和黑盒测试各有自己的优缺点，构成互补关系，在规划测试时需要把白盒测试与黑盒测试结合起来。表 6-1 给出了白盒测试与黑盒测试两类方法的对比。

表 6-1 白盒测试与黑盒测试两类方法的对比

		白盒测试	黑盒测试
测试规划		根据程序的内部结构，如语句的控制结构，模块间的控制结构及内部数据结构等进行测试	根据用户的规格说明，即针对命令、信息、报表等用户界面及体现它们的输入数据与输出数据之间的对应关系，特别是针对功能进行测试
特点	优点	能够对程序内部的特定部位进行覆盖测试	能站在用户的立场上进行测试
	缺点	无法检验程序的外部特性 无法对未实现规格说明的程序内部欠缺部分进行测试	不能测试程序内部特定部位 如果规格说明有误，则无法发现
方法举例		语句覆盖 判定覆盖 条件覆盖 判定一条件覆盖 基本路径覆盖 循环覆盖 模块接口测试	基于图的测试 等价类划分 边值分析 比较测试



6.2 软件测试的策略



6.2.1 单元测试

单元测试（Unit Testing），也称模块测试（Module Testing），其测试的主要目的是检查模块内部的错误。因此，测试方法应以白盒测试法为主。它解决 5 方面的问题：模块接口、局部数据结构、重要的执行路径、边界条件和错误处理。

1. 单元测试的内容

（1）模块接口

模块接口测试主要检查数据能否正确地通过模块。

软件单元作为一个独立的模块，同时又作为软件系统的一个组成部分，它和系统中的其他模块之间存在着信息交换，需要测试的单元要从其他模块输入信息，同时又向其他模块输出信息。因此测试信息能否正确地输入和输出待测试模块是整个单元测试的基础和前提。针对单元接口测试，Myers 提出，测试内容应该包括下列主要因素：

- ◎ 待测试单元的实参的格式是否与形参的格式一致？
- ◎ 待测试单元的实参的数据类型是否与形参的数据类型匹配？
- ◎ 调用其他单元的实参个数是否与被调用单元的形参的个数相同？
- ◎ 调用其他单元的实参数据类型是否与被调用单元的形参的数据类型相同？
- ◎ 传送给另一个被调用模块的变元，其单位是否与参数的单位一致？



- ◎ 调用库函数时，实参的个数、数据类型和顺序是否与该函数的形参表一致？
- ◎ 在模块有多个入口的情况下，是否有引用与当前入口无关的参数？
- ◎ 是否修改了只读的参数？
- ◎ 各个单元对系统中的全局变量定义和使用是否一致？
- ◎ 有没有把常数当做变量来传送？

当一个模块执行外部的输入/输出操作时，Myers 提出，还需要考虑进行以下附加的接口测试：

- ◎ 文件属性是否正确？
- ◎ OPEN 语句是否正确？
- ◎ 格式说明与输入、输出语句给出的信息是否一致？
- ◎ 缓冲区的大小是否与记录的大小匹配？
- ◎ 是否所有的文件在使用前均已打开了？
- ◎ 对文件结束条件的判断和处理是否正确？
- ◎ 对输入、输出错误的处理是否正确？
- ◎ 有没有输出信息的征文错误？

(2) 局部数据结构

在模块工作过程中，必须测试其内部的数据能否保持完整性，包括内部数据的内容、形式及相互关系不发生错误。应该说，模块的局部数据结构是经常发生错误的错误源。对于局部数据结构，应该在单元测试中注意发现以下几类错误：

- ◎ 不正确的或不一致的类型说明；
- ◎ 错误的变量名，如拼写错或缩写错；
- ◎ 不相容的数据类型；
- ◎ 下溢、上溢或地址错误。

除局部数据结构外，在单元测试中还应弄清楚全程数据对模块的影响。

(3) 重要的执行路径

重要模块要进行基本路径测试，仔细选择测试路径是单元测试的一项基本任务。测试用例必须能够发现由于计算错误、不正确的判定或不正常的控制流而产生的错误。常见的错误有：

- ◎ 误解的或不正确的算术优先级；
- ◎ 混合模式的运算；
- ◎ 精度不够精确；
- ◎ 表达式的不准确符号表示。

针对判定和条件覆盖，测试用例还应能够发现如下错误：

- ◎ 不同数据类型的比较；
- ◎ 不正确的逻辑操作或优先级；
- ◎ 应当相等的地方由于精度的错误而不能相等；
- ◎ 不正确的判定或不正确的变量；
- ◎ 不正常的或不存在的循环终止；

- ◎ 当遇到分支循环时不能退出；
- ◎ 不适当地修改循环变量。

(4) 边界条件

程序最容易在边界上出错，如输入/输出数据的等价类边界，选择条件和循环条件的边界，复杂数据结构的边界等都应进行测试。

(5) 错误处理

测试错误处理的要点是模块在工作中发生了错误，其中的错误处理设施是否有效。

程序运行中出现了异常现象并不奇怪，良好的设计应该预先估计到投入运行后可能发生的错误，并给出相应的处理措施，使得用户不至于束手无策。检验程序中的错误处理这一问题，可能出现的情况有：

- ◎ 对运行发生的错误描述难以理解；
- ◎ 所报告的错误与实际遇到的错误不一致；
- ◎ 出错后，在错误处理之前就引起了系统干预；
- ◎ 例外条件的处理不正确；
- ◎ 提供的错误信息不足，以致无法找到出错的原因。

这 5 个方面问题的提出，使得用户必须认真考虑：如何设计测试用例，以使模块测试能够高效率地发现其中的错误，这是非常关键的问题。

2. 单元测试步骤

由于被测试的模块往往不是独立的程序，它处于整个软件结构的某一层位置上，被其他模块调用或调用其他模块，其本身不能单独运行，因此在单元测试时，需要为被测试模块设计若干辅助测试模块。辅助模块有两种。一种是驱动模块（Driver），用来模拟主程序或者调用模块的功能，向被测模块传递数据，接收、打印从被测模块返回的数据。一般只设计一个驱动模块。另一种是桩模块（Stub），用来模拟那些由被测模块调用的下属模块的功能。可以设计一个或者多个桩模块，以便更好地对下属模块进行模拟。单元测试的环境如图 6-5 所示。

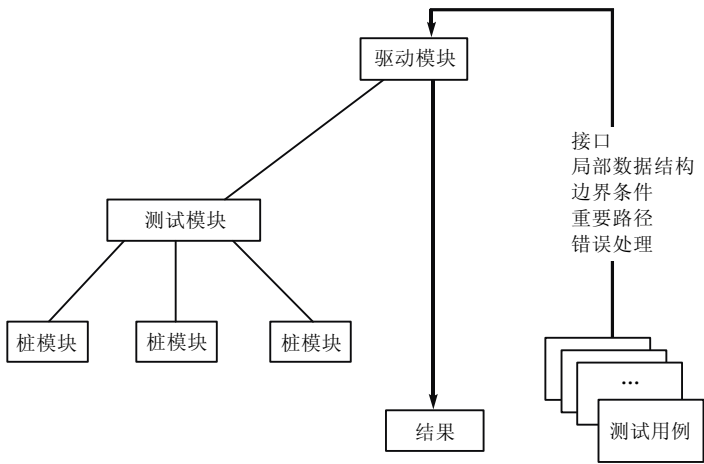


图 6-5 单元测试的环境



由于驱动模块是模拟主程序或者调用模块的功能，处于被测试模块的上层，所以驱动模块只需要模拟向被测模块传递数据，接收、打印从被测模块返回的数据的功能即可，较容易实现。而桩模块用于模拟那些由被测模块调用的下属模块的功能，由于下属模块往往不止一个，也不止一层，由于模块接口的复杂性，桩模块很难模拟各下层模块之间的调用关系，同时为了模拟下层模块的不同功能，需要编写多个桩模块，而这些桩模块所模拟的功能是否正确，也很难进行验证。所以，驱动模块的设计要比桩模块容易得多。

驱动模块和桩模块都是额外开销，这两种模块虽然在单元测试中必须编写出来，但却不作为最终的软件产品提供给用户。如果驱动模块和桩模块很简单的话，那么开销相对较低，然而，使用“简单”的模块是不可能进行足够的单元测试的，模块间接口的全面检查要推迟到集成测试时进行。

6.2.2 集成测试

在软件测试的过程中经常会遇到这样的情形：系统中的每个模块经过单元测试后都可以正常工作了，但是将这些模块组装集成在一起后却无法正常工作。出现这一情况的主要原因是模块之间的相互调用，以及相互之间的数据传送为系统的运行引入了新的问题。例如，全局数据的访问或共享出现错误，模块调用时不正确的参数传递等。为了解决这个问题，需要在每个模块完成单元测试以后，按照设计时画出的结构图，把它们连接起来进行测试，即集成测试。集成测试（Integrated Testing）是指在单元测试的基础上，将所有模块按照设计要求组装成一个完整的系统而进行的测试，也称为联合测试或组装测试。重点测试模块的接口部分，需设计测试过程所使用的驱动模块或桩模块。测试方法以黑盒测试法为主。

实践表明，一些模块能够单独地工作，并不能保证连接起来后也能正常工作。程序在某些局部反映不出的问题，在全局上很可能暴露出来，影响功能的发挥。组装测试有两种不同的方法：非渐增式测试和渐增式测试。

1. 非渐增式测试

非渐增式测试方法采用一步到位的方法来构造测试：首先对每个模块分别进行单元测试，然后再把所有的模块按设计要求组装在一起进行测试，如图 6-6（a）所示，它由 6 个模块构成。在进行单元测试时，根据它们在结构图中的地位，对模块 B 和 D 配备了驱动模块和桩模块；对 C、E 和 F 只配备了驱动模块；对主模块 A，由于它处在结构图的顶端，无其他模块调用它，因此仅为它配备了 3 个桩模块，以模拟它调用的 3 个模块 B、C 和 D，如图 6-6（b）～（g）所示。在分别进行单元测试以后，再按图 6-6（a）的结构形式连接起来，进行组装测试。

非渐增式测试方法将所有的模块一次连接起来，简单、易行，节省机时，但测试过程中难于查错，发现错误也很难定位，测试效率低。

2. 渐增式测试

渐增式测试方法与非渐增式测试方法不同，它的集成是逐步实现的，组装测试也是逐步完成的。也可以说，它把单元测试与组装测试结合起来进行。该测试逐个把未经过测试的模块组装到已经测试过的模块上去，进行组装测试。每加入一个新模块，就进行一次集成的测

试，重复此过程，直至程序组装完毕。按组装次序不同，渐增式组装测试有多种方案。

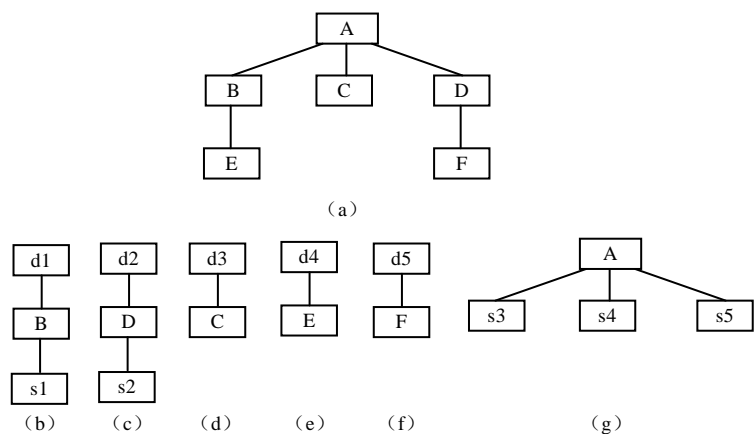


图 6-6 非渐增式的组装测试

(1) 自顶向下

本方法不需要编写驱动模块，只需要编写桩模块。其逐步集成和逐步测试工作按结构图的顺序自上而下地进行，即模块集成的顺序是，首先集成主控模块（主程序），然后按照控制层次结构向下进行集成。从属于主控模块的模块按深度优先策略（纵向）或者广度优先策略（横向）集成到结构中去。

深度优先的集成首先集成结构中一个主控路径下的所有模块。主控路径的选择是任意的，可以先选择左边的，然后中间的，直至右边的。广度优先的集成首先沿着水平方向进行，把每一层中所有直接隶属于上一层的模块集中起来，直至底层。

集成的整个过程如下。

- ① 主控模块作为测试驱动器。
- ② 根据集成的策略（深度或广度），一次一个下层模块被替换为真正的模块。
- ③ 在每个模块被集成时，都必须进行单元测试。
- ④ 回到第②步重复进行，直至整个系统结构被集成完成。

图 6-7 所示为自顶向下以深度优先策略组装模块的例子，其中 s_i 模块代表桩模块。

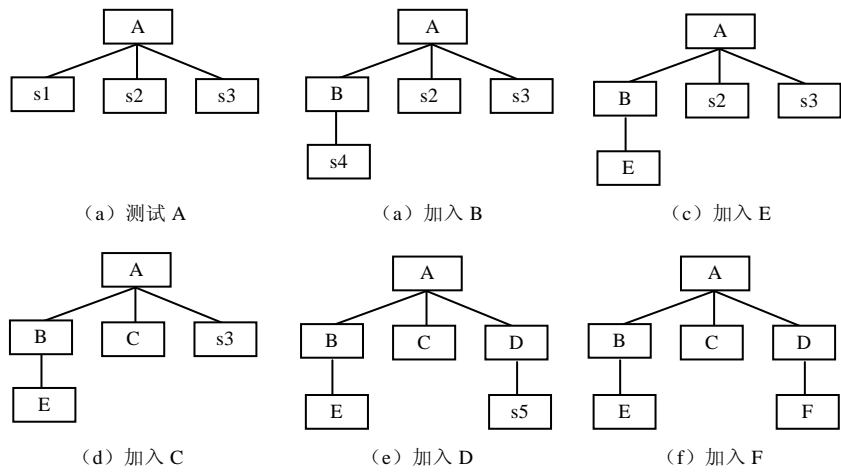


图 6-7 采用深度优先策略自顶向下结合模块的过程



自顶向下测试的优点是：能较早地发现高层模块接口、控制等方面的问题；初期的程序概貌可让人们较早地看到程序的主功能，增强开发人员的信心。其缺点是：桩模块不可能提供完整的信息，因此把许多测试推迟到用实际模块代替桩模块之后；需要设计较多的桩模块，测试开销大；早期不能并行工作，不能充分利用人力。

(2) 自底向上

本方法只需编写驱动模块，不需要编写桩模块。其逐步集成和逐步测试的工作按结构图自下而上进行。具体步骤如下。

- ① 把底层模块组合成实现一个个特定子功能的族。
- ② 为每个族编写一个驱动模块，以协调测试用例的输入和测试结果的输出。
- ③ 对模块族进行测试。
- ④ 按软件结构图依次向上扩展，用实际模块替换驱动模块，形成一个个更大的族。
- ⑤ 重复②~④步，直至软件系统全部测试完毕。

图 6-8 用同一实例演示了这一过程。

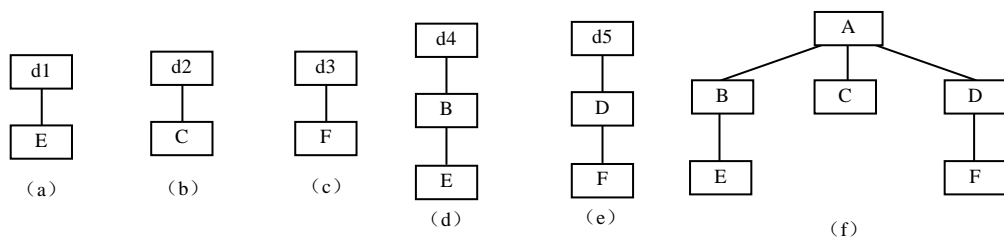


图 6-8 自底向上渐增式测试

自底向上测试的优点是：随着上移，驱动模块逐步减少，测试开销减小；比较容易设计测试用例；早期可以并行工作；下层模块的错误能较早发现。其缺点是：系统整体功能最后才能看到；上层模块错误发现得晚，而上层模块的问题是全局性的问题，影响范围较大。

(3) 混合增值

自顶向下渐增式测试和自底向上渐增式测试各有利弊，在实际应用时，应根据软件的特点、任务的进度，安排选择合适的方法。常见的混合增值方案有：

- ◎ 衍变的自顶而下——先自底而上集成子系统，再自顶而下集成总系统；
- ◎ 自底而上一自顶而下增值——对含有读操作的子系统采用自底而上方式，对含有写操作的子系统采用自顶而下方式。
- ◎ 回归测试——在回归测试中采用自底而上方式，对其余部分（尤其是对修改过的子系统）则采用自顶而下方式。

6.2.3 确认测试

组装测试完成以后，分散开发的模块被连接起来，构成完整的程序。其中各模块之间接口存在的种种问题都已消除，此时可进行测试工作的最后部分——确认测试（Validation Testing），又称为有效性测试或合格性测试（Qualification Testing）。其任务是验证系统的功

能、性能等特性是否符合需求规格说明。

确认测试阶段包括确认测试与软件配置审查两项工作。

1. 确认测试

确认测试一般是在模拟的环境（或开发环境下）下运用黑盒法，验证软件特性是否与需求符合。需要首先制定测试计划，确定测试步骤，设计测试用例；测试用例应选用实际运用的数据；测试结束后，应该写出测试分析报告。

经过确认测试后，可能有以下两种情况：

◎ 经过检验的软件功能、性能及其他要求均已满足需求规格说明书的规定，因而可能被认为是合格的软件；

◎ 经过检验发现，软件功能与需求规格说明书有相当的偏离，得到一个各项缺陷清单。

对于第二种情况，要对这样的错误进行修改，工作量非常大，往往很难在交付期以前把发现的问题纠正过来。这就需要开发部门和用户进行协商，找出解决的办法。

2. 软件配置审查

配置审查（Configuration Review）有时也称为配置审计（Configuration Audit），是确认过程的重要环节。所谓的软件配置，是指软件工程过程中所产生的所有信息项，包括：文档、报告、程序、表格、数据等。随着软件工程过程的进展，软件配置项（Software Configuration Item, SCI）快速增加和变化。

软件配置审查，应复查 SCI 是否齐全，检查软件的所有文档资料的完整性和正确性。如果发现遗漏和错误，应补充和改正。同时要编排好目录，为以后的软件维护工作奠定基础。

经过确认测试，得到测试报告，通过软件配置审查得到软件配置情况。两种测试的结果都要经过管理机构裁决后，再通过专家鉴定会的评审。

6.2.4 系统测试

软件只是计算机系统的—个组成部分，软件开发完成以后，最终还要和系统中的其他部分（如计算机硬件、外部设备、某些支持软件、数据等）集成起来，在投入运行以前完成系统测试，以确保各组成部分不仅能单独地受到检验，而且在系统各部分协调工作的环境下也能正常工作。尽管每一个检验都有着特定的目标，然而所有的检验工作都要验证系统中的每个部分均已得到正确的集成，并能完成指定的功能。以下简要地说明集中系统测试。

1. 功能测试

功能测试又称正确性测试，它检查软件的功能是否符合需求规格说明书。由于正确性是软件最重要的质量因素，所以其测试也最重要。

测试的基本方法是，构造一些合理输入，检查是否得到期望的输出。这是一种枚举的方法。倘若枚举空间是无限的，则其关键在于寻找等价区间。还有一种有效的测试方法是边界值测试。

2. 性能测试

性能测试用来测试软件在集成系统中的运行性能，特别针对实时嵌入系统。性能测试可



以在测试过程的任意阶段进行，但只有当整个系统的所有成分都集成到一起后，才能检查一个系统的真正性能。这种测试常常与强度测试结合起来进行。为记录性能，需要在系统中安装必要的测量仪表或度量性能用二进制软件（或程序段）。

3. 安全测试

安全测试的目的在于验证安装在系统内的保护机制能够在实际应用中保护系统使之不受非法侵入的干扰。系统的安全测试要设置一些测试用例，试图突破系统的安全保密措施，检验系统是否有安全保密的漏洞。

4. 恢复测试

操作系统、数据库管理系统等都有恢复机制，即当系统受到某些外部事故的破坏时，能够重新恢复正常工作。恢复测试通过各种手段，强制性地使软件出错，不能正常工作，进而检验系统的恢复能力。如果系统恢复是自动的（由系统本身完成），则应检验：重新初始化，检验点设置机构，数据恢复以及重新启动是否正确；如果这一恢复需要人为干预，则应考虑平均修复时间是否在限定的范围以内。

5. 强度测试

强度测试主要在一些极限条件下，检查软件系统的运行情况。例如，超常数量的输入数据、超常数量的用户、超常数量的网络连接，显然，这样的测试对于了解软件系统性能和可靠性、健壮性具有十分重要的意义。强度测试可以先根据所开发的软件系统所面临的一些运行强度方面的挑战设计出相应的测试用例，然后通过使用这些测试用例，了解检查软件系统在这些极端情况下是否能够正常运行。

6. 文档测试

文档测试主要检查文档的正确性、完备性和可理解性。正确性是指不要把软件的功能和操作写错，也不允许文档内容前后矛盾。完备性是指文档不可以“虎头蛇尾”，更不许漏掉关键内容。可理解性是指文档要让大众用户看得懂，能理解。

总的来说，系统测试是一项比较灵活的工作，对测试人员有较高的要求，既要了解用户的环境和系统的使用，又要具有从事各类测试的经验和丰富的软件知识。参加人员包括：有经验的系统测试专家、用户代表、软件系统的分析员或设计员。

6.2.5 α 测试和 β 测试

如前所述，即使经过一系列的严格测试，软件开发人员也不可能发现并排除软件系统中所有潜在的错误和缺陷。导致这一情况的一个重要原因是，软件开发人员不可能完全预见到用户使用软件系统的所有情况。例如，用户可能使用一组开发人员意想不到的输入数据来使用软件系统，导致系统中一些未被发现的错误或缺陷暴露出来。因此，在用户参与的情况下进行软件测试是非常重要的，它可以确认软件系统在功能和性能上是否能够满足用户的需要，并最终决定用户对该软件系统的认可程度。

α 测试是指邀请软件用户与软件开发人员一起，在开发场地对软件系统进行测试，其测

试环境要尽量模拟软件系统投入使用后的实际运行环境。在测试过程中,软件系统出现的错误或使用过程中遇到的问题,以及用户提出的修改要求,均要由开发人员完整、如实地记录下来,作为对软件系统进行修改的依据。 α 测试的整个过程是在受控环境下,由开发人员和用户共同参与完成的。 α 测试的目的是评价软件的 FLURPS。FLURPS 包含的测试项目如下:

- ◎ Function Testing——功能测试;
- ◎ Local Area Testing——局域化测试;
- ◎ Usability Testing——可使用性测试;
- ◎ Reliability Testing——可靠性测试;
- ◎ Performance Testing——性能测试;
- ◎ Supportability Testing——可支持性测试。

β 测试是指由软件产品的全部或部分用户在实际使用环境下进行的测试。整个测试活动是在用户的独立操作下完成的,没有软件开发人员的参与。 β 测试是投入市场前由支持软件预发行的客户对 FLURPS 进行的测试,其主要目的是测试系统的可支持性。 β 测试的涉及面最广,最能反映用户的真实愿望,但花费的时间最长,不好控制。一般,软件公司与 β 测试人员之间有一种互利的协议,即 β 测试人员无偿地为软件公司作测试,定期递交测试报告,提出批评与建议;而软件公司将向 β 测试人员免费赠送或者以很大的优惠价格提供软件的正式版本。

6.2.6 综合测试策略

软件测试是保证软件可靠性的主要手段,也是软件开发过程中最艰巨、最繁杂的任务。软件测试方案是测试阶段的关键技术问题,其基本目标是选择最少量的高效测试用例,从而尽可能多地发现软件中的问题。因此,无论哪一个测试阶段,都应该采用综合测试策略,才能够实现测试的目标。

一般应该先进行静态测试,再考虑动态测试。

1. 单元测试

通常应该先进行“人工走查”,再以白盒法为主,辅以黑盒法进行动态测试。使用白盒法时,只需要选择一种覆盖标准,而使用黑盒法时,应该采用多种方法。

2. 组装测试

关键是要按照一定的原则,选择组装模块的方案(次序),然后再使用黑盒法进行测试。在测试过程中,如果发现有问题较多的模块,需要进行回归测试时,再采用白盒法。

3. 确认测试、系统测试

应该以黑盒法为主。在确认测试中进行软件配置复查,主要是静态测试。



6.3 软件调试

软件调试是指在软件测试完成后,对在测试过程中发现的错误加以修改,以保证软件运



行的正确性和可靠性。显然，软件调试活动主要分为以下三个部分。

- ① 确定软件系统出现错误的准确位置；
- ② 对发现的错误进行纠正和修改；
- ③ 对修改以后的模块或整个系统重新进行测试。

重复以上 3 个步骤，直到整个软件系统能够得到正确的运行结果。



6.3.1 软件调试过程

软件调试是在完成软件测试以后，修改和纠正软件系统的错误的过程。软件调试的过程具体如下。

- ① 从软件测试过程中发现的错误的表现形式入手，确定软件系统出现错误的原因。
- ② 对软件系统进行细致研究，确定错误发生的位置。
- ③ 修改软件系统的设计和编码，排除或纠正发现的错误。
- ④ 对修改以后的软件系统进行重复测试，以确保对错误的排除和纠正没有引入新的错误。
- ⑤ 如果发现针对错误进行的修改没有效果甚至引入了新的错误，那么需要根据实际情况撤销此次修改或者对新出现的错误进行修改。

不断重复以上过程，直至在软件测试中发现的错误都被消除，并且没有引入新的错误为止。

在整个软件系统开发中，调试工作是一个漫长而艰难的过程，软件开发人员的技术水平乃至心理因素对软件调试的效率和质量都有很大的影响。从技术角度来看，软件调试的困难主要有以下几个方面。

- ① 人为因素导致的错误不易被确定和追踪。
- ② 当一个错误被纠正时，可能会引入新的错误。
- ③ 在软件系统中，错误发生的外部位置与其内在原因所处的位置可能相差甚远。
- ④ 在分布式处理环境中，错误的发生是由若干个 CPU 执行的任务共同的，因而错误的准确定位十分困难。
- ⑤ 错误是由难以精确再现的外部状态或事件所引起的。

软件调试是一项十分艰巨的工作，要在规模庞大的软件系统中准确地确定错误发生的原因和位置，并、纠正相应的错误，这需要有良好的调试策略。



6.3.2 软件调试策略

软件调试工作的关键是采用恰当的调试策略，发现并纠正软件系统中发生的错误，常见的软件调试策略包括：试探法、归纳法、演绎法、回溯法和分查找法。下面具体介绍一下这几种调试策略的基本思想和特点。

1. 试探法调试

试探法是一种比较原始的调试策略。它的基本思想是，通过分析软件系统运行过程中数据信息、中间结果的变化情况，来查找错误发生的原因、确定错误发生的位置。例如，通过输出寄存器、内存单元的内容，在程序中的恰当位置插入若干条输出语句等方法，来获取程序运行过程中的大量现场信息，从中发现出错的线索。

使用试探法来获取错误信息具有很大的盲目性，那样需要耗费大量的时间和精力，因此该方法具有较低的调试效率，并且由于采用的调试技术十分原始，使其只适用于对结构比较简单的小规模系统的调试，而对于复杂的大型系统却无能为力。使用试探法的典型方式包括：输出作为程序中间结果的相关数据的值，在程序中添加必要的打印语句，使用自动调试工具。在许多集成开发环境（IDE）中都包含有相应的调试工具，例如，设置程序执行的断点、程序单步执行等。这些调试工具的使用可以有效地帮助开发人员完成对软件系统的调试工作。

2. 归纳法调试

归纳是一种由特殊到一般的逻辑推理方法。归纳法调试是根据软件测试所取得的错误结果的个别数据，分析出可能的错误线索，研究出错规律和错误之间的线索关系，由此确定错误发生的原因和位置。归纳法调试的基本思想是：从一些个别的错误线索着手，通过分析这些线索之间的关系而发现错误。

如图 6-9 所示，归纳法调试的具体实施步骤如下。

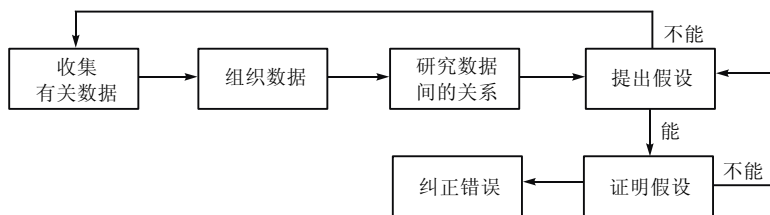


图 6-9 归纳法调试的步骤

第 1 步：收集有关数据

对所有已经知道的测试用例和程序运行结果进行收集、汇总，其中不仅要包括那些出错的运行结果，也要包括那些不产生错误结果的测试数据，这些数据将为发现错误提供宝贵的线索。

第 2 步：整理分析有关数据

对第 1 步收集的有关数据进行组织、整理，并在此基础上对其进行细致的分析，从中发现错误发生的线索和规律。

第 3 步：提出假设

研究分析测试结果数据之间的关系，力求寻找出其中的联系和规律，进而提出一个或多个关于出错原因的假设。如果无法提出相应的假设，则回到第 1 步，补充收集更多的测试数据；如果可以提出多个假设，则选择其中可能性最大者。

第 4 步：证明假设

在假设提出以后，证明假设的合理性对软件调试是十分重要的。证明假设是指将假设与原始的测试数据进行比较，如果假设能够完全解释所有的调试结果，那么该假设便得到了证明；反之，该假设就是不合理的，需要重新提出新的假设。



3. 演绎法调试

演绎是一种由一般到特殊的逻辑推理方法。演绎法调试是指根据已有的测试数据，设想所有可能的出错原因，然后通过测试逐一排除不正确、不可能的出错原因，直到最后证明剩余的错误原因的确是软件系统发生错误的根源。

具体来说，演绎法调试主要包括如图 6-10 所示的三个步骤。

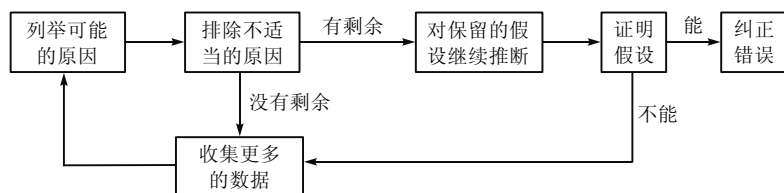


图 6-10 演绎法调试的步骤

第 1 步：设想所有可能的出错原因

根据已有的测试用例和测试结果数据，设想、推测出软件系统之所以发生相关错误的所有可能的原因。

第 2 步：排除不可能的出错原因

针对第 1 步中获得的各种可能的出错原因，通过软件测试，逐一排除其中与测试结果有矛盾（即不可能）的出错原因。

第 3 步：验证可能的出错原因

针对经过第 2 步的排除后剩余的那些可能的出错原因，使用软件的测试结果，对其合理性进行验证，并进一步确定错误发生的位置。

4. 回溯法调试

回溯法是指从软件系统中发现错误的位置开始，沿着程序的控制流程往回追踪程序代码，直至找到错误发生的位置或范围为止。

回溯法对于规模较小的软件系统而言是一种比较有效的调试策略，它能够将错误的范围缩小到程序中的某一个较小部分，为错误的精确定位提供了方便。但是，随着程序规模的不断扩大，进行回溯的流程路径的数目将会急剧增加，使得进行流程的完全回溯变得不现实。

5. 对分查找法调试

如果已经知道某些变量在程序中若干关键点正确值，则可以在程序中间的某个恰当位置插入赋值语句或输入语句为这些变量赋予正确的值，然后再检查程序的运行结果。如果在插入点以后的运行正确，那么错误一定发生在插入点的前半部分；反之，则错误一定发生在插入点的后半部分。对于程序中有错误的部分再重复使用该方法，直至把错误的范围缩小到容易诊断的区域为止。



6.4 面向对象的测试

面向对象技术是一种全新的软件开发技术，正逐渐代替曾被广泛使用的面向过程开发方

法,被看成是解决软件危机的新兴技术。面向对象技术产生更好的系统结构,更规范的编程风格,极大地优化了数据使用的安全性,提高了程序代码的重用性。

尽管面向对象技术的基本思想保证软件有更高的质量,但无论采用什么样的编程技术,编程人员的错误都是不可避免的,而且由于面向对象技术开发的软件代码重用率高,更需要严格测试,避免错误的繁衍。因此,软件测试并没有因为面向对象技术的兴起而丧失掉它的重要性。

但由于面向对象的软件开发方法与传统的软件开发模式有着很大的不同,对每个开发阶段都有不同以往的要求和结果,已经不可能用功能细化的观点来检测面向对象分析和设计的结果。

面向对象的测试,既要使用许多传统的成熟的软件测试方法和技术,也有其不同的特点,主要反映在测试对象和内容的不同。但无论如何,测试的目标是不会改变的。我们必须注意到,面向对象的测试比传统的测试更加复杂和困难。

近年来,随着面向对象软件开发方法应用的更加广泛和研究的不断深入,面向对象软件测试已成为软件工程领域的一个重要研究课题。

6.4.1 面向对象测试的特点

传统的观点认为,测试要在编码之后才进行,主要测试的对象是程序代码。而面向对象的测试认为,测试是一种被应用在开发过程中不同阶段点的活动,贯穿于软件开发的全过程,它与开发过程的每个阶段都密切相关。

传统测试模式与面向对象的测试模式的最主要的区别在于,面向对象的测试更关注于对象而不是完成输入/输出的单一功能,这样的话,测试可以在分析与设计阶段就先行介入,使得测试能够更好地配合软件生产过程并为之服务。与传统测试模式相比,面向对象测试的优点如下。

- ① 能更早地定义出测试用例,早期介入进行测试可以降低成本。
- ② 尽早编写系统测试用例,以便于开发人员与测试人员对系统需求的理解保持一致。
- ③ 面向对象的测试模式更注重软件的实质。

由于面向对象程序的结构不再是传统的功能模块结构,作为一个整体,原有集成测试所要求的逐步将开发的模块搭建在一起进行测试的方法已成为不可能。

面向对象的测试与传统的测试方法比较,其特点反映在以下两个方面。

① 强调需求或设计的测试,也就是说,将测试工作提前到编码前,而且以需求和设计阶段的测试为主,在软件开发的早期就开始测试工作,能够保证需求和设计的高质量,可以有效地防止和减少错误的蔓延。通常采用以下两种方式进行。

◎ 在没有代码的情况下进行测试,主要是验证和确认规格说明的有效性和正确性,一般采用静态走查和动态的场景模拟等方法。

◎ 在有代码的情况下进行测试,以规格说明为依据,验证代码的正确性。

② 在传统测试方法的基础上,根据面向对象的主要特性,需要改变测试策略和方法。

例如,封装是对数据的隐蔽,减少了对数据非法操作,可简化该类测试。继承性提高了



代码复用性，但错误也会以同样方式被复用。多态性提供强大的处理能力，但也增加了测试的复杂性。

6.4.2 面向对象测试的类型

面向对象的开发模型突破了传统的瀑布模型，将开发过程分为面向对象分析（OOA）、面向对象设计（OOD）和面向对象编程（OOP）三个阶段。分析阶段产生整个问题空间的抽象描述，在此基础上，进一步归纳出适用于面向对象编程语言的类和类结构，最后形成代码。由于面向对象的特点，采用这种开发模型能有效地将分析设计的文本或图表代码化，不断适应用户需求的变动。

根据面向对象的软件开发过程的特点，提出面向对象的软件测试技术，建立一种在整个软件开发过程中不断测试的测试模型，包括分析与设计模型测试、类测试、对象交互测试、类层次结构测试、面向对象系统测试几部分。

1. 分析与设计模型测试

采用正式技术评审的方法，检查分析与设计模型的正确性、完整性和一致性。按照测试对象的不同，通常模型测试方法包括：用例场景测试，系统原型走查，需求模型一致性检查，分析模型的检查和走查。

测试的主要内容有：对确定的对象的测试，对确定的结构的测试，对确定的主题的测试，对定义的属性和实例关联的测试，对定义的服务和消息关联的测试。

2. 类测试

面向对象软件产品的基本组成单位是类。从宏观上来看，面向对象软件是各个类之间的相互作用。在面向对象系统中，系统的基本构造模块是封装了的数据和方法的类和对象，而不再是一个个能完成特定功能的功能模块。

类测试对应于传统测试中的单元测试。类的测试是验证类的实现与类的说明是否一致的活动。类测试包括：类属性的测试、类操作的测试、可能状态下对象的测试。测试中要特别注意：不能“孤立”进行测试，操作测试应该包括其可能被调用的各种情况。对象中的数据和类是一个有机的整体，在测试过程中不能仅检查输入数据产生的输出结果是否与预期的吻合，还要考虑对象的状态。

假设在进行模型测试时，已经对类的完整性说明进行了测试。此时，类测试的内容主要是确保一个类的代码能够完全满足类的说明所描述的要求。对一个类进行测试以确保它只做规定的事情。

类测试的方法有代码检查和执行测试用例两种。如果类的实现正确，那么类的每一个实例的行为也应该是正确的。

3. 对象交互测试

对象交互测试用于代替传统测试方法中的集成测试。对类进行交互测试，以确定它们能否在一起共同工作。交互测试的重点是，确保那些已经单独测试过的类的对象相互之间能够

正确地传送消息。

面向对象的交互（集成）测试能够检测出相对独立的类（单元）测试无法检测出的那些在类相互作用时才会产生的错误。基于单元测试对成员函数行为正确性的保证，交互测试只关注系统的结构和内部的相互作用。

传统的自顶向下和自底向上的集成策略对面向对象的测试集成是无意义的，因为，面向对象的软件没有层次控制结构，而且，一次集成一个操作到类中（传统的增量集成方法）经常是不可能的。

对面向对象的集成测试必须采用新的方法。通常有两种不同的策略。

① 基于线程的测试（Thread-Based Testing），集成对系统的一个输入或事件所需的一组类，每个线程被集成并分别测试。

② 基于使用的测试（Use-Based Testing），首先，测试独立类（几乎不使用服务器的类）而开始构造系统；然后，测试下一层的依赖类（使用独立类的类），通过依赖类层次的测试序列逐步构造完整的系统。

在进行交互测试时，需要注意以下问题。

① 类间的继承性可能给测试带来新的困难。继承性的含义是，一个类中定义的操作和属性可由另一个类继承，并且可以在继承的位置执行。因此继承性层次的测试需要更彻底的测试方法，必须知道系统中的操作如何出现。继承性较高层次测试的操作在较低层次中测试并不总是成立的。

② 发送一个消息给自身。这样的消息或许只与一个派生类相关，因此在这种情况下，测试抽象类没有价值，为了弄清发送给自身的信息系列，在类层次中需要从上到下、从下到上的工作，这种测试称为正向、逆向测试法。

4. 系统（子系统）测试

通过类测试和交互测试，仅能保证软件开发的功能得以实现，但不能确认在实际运行时，它是否满足用户的需要，是否大量存在实际使用条件下会被诱发产生错误的隐患。还必须测试系统或独立子系统，确保系统无明显故障，并满足用户需求。

系统测试应该尽量搭建与用户实际使用环境相同的测试平台，应该保证被测系统的完整性，对临时没有的系统设备部件，也应有相应的模拟手段。系统测试时，应该参考 OOA 分析的结果，对应描述的对象、属性和各种服务，检测软件是否能够完全“再现”问题空间。系统测试不仅是检测软件的整体行为表现，从另一个侧面看，也是对软件开发设计的再确认。

系统测试包括以下内容。

- ◎ 功能测试。测试是否满足开发要求，是否能够提供设计所描述的功能，是否用户的需求都得到满足。功能测试是系统测试最常用和必需的测试，通常还会以正式的软件说明书为测试标准。
- ◎ 压力测试。测试系统的能力最高实际限度，即软件在一些超负荷条件下功能实现情况。例如，要求软件某一行为的大量重复、输入大量的数据或大数值数据、对数据库进行大量复杂的查询等。
- ◎ 安全测试。验证安装在系统内的保护机构确实能够对系统进行保护，使之不受各种非常的干扰。安全测试时需要设计一些测试用例试图突破系统的安全保密措施，检



验系统是否有安全保密的漏洞。

- ◎ 性能测试。测试软件的运行性能。这种测试常与强度测试结合进行，需要事先对被测软件提出性能指标，如传输连接的最长时限、传输的错误率、计算的精度、记录的精度、响应的时限和恢复时限等。
- ◎ 恢复测试。采用人工的干扰使软件出错，中断使用，检测系统的恢复能力，特别是通信系统。恢复测试时，应该参考性能测试的相关测试指标。
- ◎ 可用性测试：测试用户是否能够满意使用，具体体现为：操作是否方便，用户界面是否友好等。

还有其他测试，如安装/卸载测试（Install/Uninstall Testing）等。

5. 验收和发布测试

验收测试：交付用户前的系统测试。

发布测试：确保系统安装软件包能够正常交付使用。



6.4.3 分析模型测试

1. 分析模型测试的重要性

（1）需求的质量影响并决定了设计的质量

在软件开发过程模型中，需求、设计和编码总是有一定的时序特性。而且，需求模型、设计模型和实现代码之间还具备解释特性，即设计解释了需求，实现代码解释了设计。

（2）需求测试可以较早地发现需求中问题

例如，不合理的项目、错误地理解了用户需求的项目，避免对成本和资源的消耗。

（3）减少需求的模糊性

用户需求是用户对待实现的系统的要求，通常以一种非正规的形式给出，具有一定的模糊性。这种模糊性如果带入到设计，甚至代码中，将可能引发几倍，甚至几十倍的错误，这必将极大地消耗系统的资源和成本。

测试实际上也是一个项目，也有需求、设计和实现，并且测试本身也会有测试（测试中的测试）。测试作为项目开发活动中的一部分，在时间上应该有明确的要求。测试计划对于测试来说是至关重要的。

UML 分析模型的每个模式，从严格意义上说都应该经过测试。实际上，通常对用例模型、类对象模型以及用例中典型场景进行测试。

2. 用例模型测试

单个用例测试采取典型应用场景的测试方法，用例模型的测试则相当于系统测试，测试的主要目标是用例模型对于用户需求的可跟踪性。

以系统的用户为主要出发点设计测试用例，通过模拟某个系统用户的行为来测试整个系统，对于该用户的服务提供情况，从而检查系统功能的完整性，用户需求可跟踪性等情况。

用例模型的测试从系统用户的角度测试系统的服务，并不关心每个测试用例所实现的功能如何，所以应该是黑盒测试。

下面以一个订货中心系统的用例模型为例说明测试用例的设计。

【例 6-2】有一个订货中心，接受客户的电话、传真、电子邮件、信件和 Web 主页表单等形式的订货请求，建立订货单。根据客户要求的发货目的地，订货中心将以最经济的方式确定一家仓库来负责向客户发货。当仓库收到订货单后，按照一定策略进行处理发货，填写订货单发货的有关信息后，将订货单返回订货中心。

(1) 根据需求分析的结果，识别出 5 个主要的系统角色（用户）：

管理者(Manager)、发货人员(Shipper)、收款人员(Toll Collector)、商务客户(Customer)、信用卡(Credit Card)

(2) 从各个角色出发，通过回答以下问题识别用例：

- ① 角色要求系统提供的功能有哪些？系统在提供这些功能时，该角色需要做什么？
- ② 角色需要创建、阅读、销毁或存储系统的哪些信息？
- ③ 系统中的哪些事件需要通知该角色？

以管理者为例。

① 管理者要求系统为他提供什么功能？管理者需要做哪些工作？

答：管理者要求系统提供以下功能：

- ◎ 接受顾客订货请求并创建订单；
- ◎ 计算订单的价钱；
- ◎ 根据订单信息选择仓库，并将订单发送给仓库；
- ◎ 查询订单货物发送情况；
- ◎ 查询客户订单付款情况；
- ◎ 评价商务结果；
- ◎ 顾客退货处理；
- ◎ 把仓库返回的订单发送到收费处；
- ◎ 商品价格更新。

管理者需要做的工作：生成订单；查询订单时输入订单号。

② 管理者需要阅读创建、销毁、更新或者存储系统的哪些信息？

答：信息包括：订单、职员（仓库人员、收费人员等）信息、顾客信息、物品条目及价格信息、仓库信息和税务信息。

③ 系统中的哪些事件需要通知管理者？

答：这些事件包括：仓库中某物品短缺以致无法满足某订单，订单数据出现错误，顾客超过期限未付款。

可见，管理者要使用系统的 10 中某个功能，因此至少可以设计出 10 个测试用例。下面以第 3 条功能“根据订单信息选择仓库，并将订单发送给该仓库”为例，说明测试用例的设计。

假设订货中心下属共有 3 个仓库，各仓库的有关信息如表 6-2 所示，管理者要决定选择哪个仓库处理订单。



表 6-2 订货中心仓库信息

仓库名称	仓库位置	存货品名及数量	订单处理客户信誉度
A	东城	G1(200),G5(100),G6(1000),G10(70),G11(90)	85
B	西城	G1(1000),G2(100),G5(550),G8(150),G10(980)	95
C	北城	G1(220),G4(300),G5(350), G7(400), G10(700)	80

订单主要信息：订单号、送货地点、货物名称及数量等。

管理者考虑将订单分配到某仓库的原则是：

- ◎ 首先仓库必须能够满足订单上的货物要求；
- ◎ 选择地理位置与发货点较近的仓库发货；
- ◎ 信誉满意度越高的客户就越应该以较高的服务质量来回报。

综合考虑上面 3 个因素，以最低的成本取得最好的收益，3 张订单的信息参见表 6-3。

表 6-3 订单信息

订单号	送货地点	货物名称及数量	客户信誉
订单 1	北城某集团公司	G1(200),G5(100), G10(40)	95
订单 2	东城某街道	G5(10), G6(5)	80
订单 3	北城某街道	G4(10)	85

根据仓库信息、订单信息及分配仓库的原则，确定有关订单的 3 个用例如下。

测试用例 1：

输入：订单 1

预期结果：选择仓库 B 来处理订单（3 个均可，大宗订单，客户信誉度高）

测试用例 2：

输入：订单 2

预期结果：选择仓库 A 来处理订单（个人订单，客户信誉一般）

测试用例 3：

输入：订单 3

预期结果：选择仓库 C 来处理订单

以上测试未触及某个具体用例，体现了用例模型测试和用例测试的区别。

3. 类的测试

类测试即传统测试中的单元测试，即验证类的实现与类的规约是否一致的活动。

类测试包括：类属性的测试、类操作的测试等。

【例 6-3】如图 6-11 所示，Date 类是一个描述日期的类，其属性为 3 个成员变量：年、月、日。在 Date 类中，操作 decrease()使 Date 类的对象改变为当前日期的前一天。PrintDate()打印日期信息。而 Date 类的 3 个成员变量所属的类是 calendarUnit 类的子类，如图 6-12 所示。在 calendarUnit 类中，也有操作 decrease()，并通过继承关系在 Day、Month、Year 三个类中重写。

设计测试用例，对 Date 类的操作 decrease()进行测试。

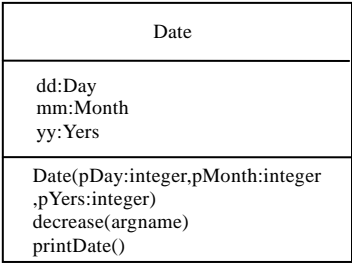


图 6-11 Date 类

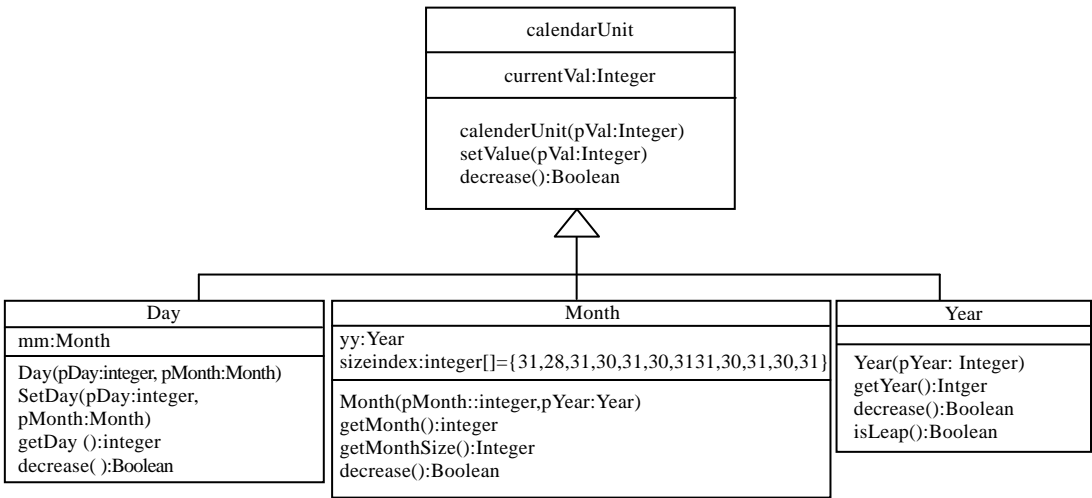


图 6-12 calendarUnit 类的继承关系

操作测试时，应该考虑其可能被调用的各种情况。对 Date 类中的 decrease 方法进行测试，在设计测试用例时，采用等价分类法。划分等价类时首先要考虑一些一般情况，如一个月中间一天的前一天，一个月第一天的前一天，一年的第一天的前一天等。除此而外，还要考虑边界及闰年、闰月等特殊情况。在前面进行的分析的基础上，划分的有效等价类和无效等价类如下：

有效等价类：

- D1=|一个月的第一天与最后一天之间|
- D2=|一个月的第一天|
- D3=|1 月 1 日|
- M1=|前一个月是 30 天|
- M2=|前一个月是 31 天|
- M3=|前一个月是 2 月|
- Y1=|非闰年|
- Y2=|闰年|
- Y3=|2005 年|



无效等价类:

D4=|<本月的第一天|

D4=|>本月的最后一天|

M4=|<1|

M5=|>12|

Y=|<0|

根据划分的有效等价类和无效等价类,设计相应的测试用例,如表 6-4 所示。注意,一个测试用例只能覆盖一个无效等价类。进行简化后,共考虑了 22 种情况。

表 6-4 测试用例

用例编号	月	日	年	预期结果
1	7	19	1998	1998 年 7 月 18 日
2	9	19	2004	2004 年 9 月 18 日
3	3	19	2000	2000 年 3 月 18 日
4	7	1	1998	1998 年 6 月 30 日
5	9	1	2004	2004 年 8 月 31 日
6	3	1	1998	1998 年 2 月 28 日
7	3	1	2004	2004 年 2 月 29 日
8	1	1	1998	1997 年 12 月 31 日
9	1	1	2004	2003 年 12 月 31 日
10	1	1	2000	1999 年 12 月 31 日
11	7	0	1998	无效输入
12	7	32	1998	无效输入
13	9	31	2004	无效输入
14	2	29	1998	无效输入
15	2	30	2004	无效输入
16	0	19	1998	无效输入
17	0	0	1998	无效输入
18	13	19	2004	无效输入
19	13	0	2004	无效输入
20	7	19	0	无效输入
21	0	19	0	无效输入
22	0	0	0	无效输入

4. 类模型的测试

类模型是分析模型的核心,它抽象出了问题域中的对象和实体,以及它们在问题域中的职责。为确保类模型的正确性和完整性,只根据问题域测试类模型。测试方法是评审会。

由于类图实际上是由类和类之间的关系组成的,评审会的检查单可根据以下两个方面的问题制定。

① 针对每个类提问:

- ◎ 该类在问题域中对应的实体(或对象)是什么?
- ◎ 履行什么职责?
- ◎ 在类图中被赋予了哪些职责?
- ◎ 该类在问题域中的职责和在类图中的职责能匹配吗?
- ◎ 该类的每个数据属性都是问题域所关心的吗?

② 针对类图中的类之间的关系提问:

- ◎ 这种类关系反映了问题域本质的关系还是为了管理类模型而引入的关系?(如果类之间的关系并不反映问题域的本质,那么这个关系的存在就值得怀疑。)
- ◎ 仔细检查每个继承关系,到底是聚集关系还是继承关系?
- ◎ 针对关联关系中的关联数目,提出一些问题结合实际场景来进行考察。



6.4.4 面向对象的测试用例

目前关于面向对象软件的测试用例的设计方法仍然在研究之中,还没有一个统一、规范的方法。Berard 在 1993 年提出了关于面向对象测试用例设计方法的一些指导性意见:

- ① 每个测试用例都应定义明确的测试目的;
- ② 每个测试用例都应定义唯一的标识,并指明与之关联的被测试类;
- ③ 每个测试用例都应定义相应的测试步骤、被测对象的状态、测试所使用的消息和操作以及测试可能产生的错误等内容。

在面向对象的测试用例的设计中,通常采用两种设计方法,即基于故障的方法和基于用例的方法。

基于故障的方法首先通过对面向对象分析模型和设计模型的理解,找出软件系统中可能存在的故障,然后以此为基础设计相应的测试用例,最后通过使用这些测试用例确定在软件系统中相应的故障是否出现。

基于用例的方法则从用户的需求出发,首先获得软件系统需要满足的需求和完成的功能,然后以此为基础设计出相应的测试用例,最后通过使用这些测试用例确定所开发的软件系统是否能够实现相应的功能、满足用户的需求。

在设计面向对象的测试用例时,可以参考使用如下的设计步骤:

- ① 确定需要被测试的类;
- ② 确定测试所采用的覆盖标准;
- ③ 确定待测试类与其他类或外部系统的相互关联;
- ④ 设计相应的测试用例,完成对该类的测试工作。



习题 6

- 6.1 等价分类法的基本思想是什么？
- 6.2 自顶而下增值与自底而上增值各有什么优点、缺点？
- 6.3 渐增式与非渐增式各有何区别？为什么通常采用渐增式？
- 6.4 什么是 α 测试和 β 测试？
- 6.5 黑盒法与白盒法的区别是什么？各自适合在什么情况下使用？
- 6.6 软件测试与其他软件开发活动相比具有什么样的特点？
- 6.7 软件测试通常包含哪几个基本步骤？
- 6.8 软件调试有哪些方法？各自有什么特点？
- 6.9 面向对象的测试与传统的测试有什么相同和不同之处？
- 6.10 面向对象的集成测试，与传统的集成测试有何区别？

第 7 章 面向对象程序设计语言的核心概念

数据封装、继承和多态性是每一种面向对象的程序设计语言必须实现的核心概念。



7.1 面向对象的目标

传统的程序设计方法是模块化（或结构化）的程序设计方法，具体步骤为：

- ① 整个软件系统功能逐步细化为多个小的功能——功能划分；
- ② 每个小的功能对应由一个模块（如函数、过程、分程序、子程序等）来实现；
- ③ 多个模块合作完成较大的功能，所有模块的合作完成整个软件系统的功能。

对于传统的程序设计，在设计和实现（编程）阶段考虑的是模块，程序本身也是由模块构成的，称之为面向模块的。

那么什么又是面向对象的呢？

下面以数据类型的概念来引入面向对象的概念。

数据类型是一个抽象的概念，包含有一组数据的定义和一组对该组数据操作的定义。数据类型分为以下三种。

- ◎ 简单数据类型：又称为内建（Built-in）类型，是语言本身提供的，如整型，包含所有的整数和对整数的操作，对于一个整数的成分（二进制数形式）是不可见的，也不可直接操作。
- ◎ 用户定义数据类型：以简单数据类型为基础，它包含的数据成分是两个简单数据类型的数据（或已经定义好的其他用户定义数据类型的数据），可以对数据成分进行直接操作。
- ◎ 抽象数据类型：在定义数据时，必须同时定义对数据的操作，它的成分（简单数据类型数据或用户定义数据类型数据）是不可见的，也不可直接操作，必须通过类型提供的操作进行访问。

在一个高级程序设计语言的程序中，对于简单数据类型和用户定义数据类型，数据的定义和操作是分开的，只是在对数据进行操作时，需要检查该操作是否符合对应的类型允许的操作（即类型检查）。

而抽象数据类型是从更一般的信息隐蔽（Information Hiding）原则派生出来的。抽象数据类型隐蔽了表示的细节，通过过程（或方法）来访问抽象数据对象。对象的表示是被保护的，防止了任何外界对它的直接操作。要对抽象数据对象进行访问，只能通过抽象数据类型中提供操作才能进行。



对于类型的使用，必须通过类型的实际例子（简称实例或实体，即传统语言中的变量或常量）的使用来体现。

数据 1
数据 2
...
数据 n
操作 1
操作 2
...
操作 m

图 7-1 对象的模型结构

面向对象语言中的对象是“将某组数据和使用该组数据的一组基本操作封装在一起而形成的一个实体”。实际上就是抽象数据类型的一个实例。如图 7-1 所示。

对象和抽象数据类型的关系，就像整型变量和整型的关系。

面向对象的基本想法就是把要构造的系统表示为对象的集合。这里所说的系统，不仅应考虑为程序和软件系统，而且应广泛解释为计算模型、CAD/CAM 中的成分模型以及人工智能中的知识表示形式等。

面向对象的思想与解决计算机软件面临的两大课题有关：一个课题是，怎样克服软件的复杂性；另一个课题是，怎样将现实世界模型在计算机中自然地表示出来（还要包括它们之间的关系）。

客观世界的问题都是由客观世界的实体及其相互间的联系构成的，把客观世界的实体称之为问题空间的对象。显然，“对象”因问题的特殊性，是不固定的，一本书可以是一个对象，一人也可以是一个对象，每一个对象都有自己的运动规律和内部状态，不同对象之间的相互作用和互相通信构成了完整的客观世界。因此，从思维模型的角度，面向对象很自然地与客观世界相对应。

从软件的角度，计算可以看做仿真。考虑一个雇员数据库，库中每个记录都代表一个雇员，可以说这是对公司的某些方面的一种仿真或模拟。同样地，一个操作系统的数据结构反映了真实世界中某些对象的状态。例如，可以用来反映这样的事实：一个磁盘正处于奇偶校验状态，或者系统中的一个设备正分配给某个特定的作业。

如果每个被仿真的实体都由一个数据结构来表示，并且将相关的操作信息封装进去，仿真将被简化，可以方便地刻画对象的内部状态和运动规律。面向对象就是这样一种适用于直观模型化的设计方法。这意味着，系统设计者从现实世界所得到的图像，或设计者头脑中形成的模型里所出现的物理图像，与构成系统的一组对象之间有近乎一对一的对应关系。这一思想非常利于实现大型的软件系统。

作为克服软件复杂性的手段，在面向对象中，利用了如下对象的性质：

- ① 将密切相关的数据和过程封装起来定义为一个实体；
- ② 定义了一个实体后，即使不知道此实体的功能是怎样实现的，也能使用它们（这一点类似于库函数，对于使用库函数的人而言，不必关心库函数的实现细节，只要知道该函数的原型，就可以正确地使用该函数）。

这相当于软件工程和程序设计方法论中的抽象化、抽象数据类型和信息隐藏等概念所具有的性质。它把系统中所有资源，如数据、模块以及系统，都看做对象。每个对象把一组数据和一组过程封装在一起，这组过程对这组数据进行处理。使用这一方法，设计人员可以依照自己的意图创建自己的对象，并将问题映射到该对象上。该方法直接、自然，可以使设计人员把主要精力放在系统一级，而对细节问题可以较少地关心。

使用对象具有信息局部化、使程序结构与设计结构相吻合的优点，有利于在完善和维护

阶段对软件进行修改,也有利于其他人(非设计人员)来清除软件错误。程序员容易确定程序的哪些部分依赖于正要修改的片断,而且正在修改的部分对其他部分影响很小。这对大型、复杂软件的维护和改进是很重要的。

面向对象设计非常注重设计方法,因为它要产生一种与现实具有自然关系的软件系统,而现实就是一种模型。实际上,用面向对象方法编程的关键是模型化。程序员的责任是构造现实的软件模型。此时,计算机的观点是不重要的,而现实生活的观点才是最重要的。

因此,可以将面向对象的目标归纳为:对试图利用计算机进行问题求解和信息处理的领域,尽量使用对象的概念,将问题空间中的现实模型映射到程序空间,由此所得到的自然性可望克服软件系统的复杂性,从而得到问题求解和信息处理的更高性能。

将一组性质和功能相同的数据归为一类,使用一个抽象数据类型来定义它们。对于相互间有联系或关系的数据,还要考虑它们之间的联系或关系。

在进一步叙述面向对象设计方法之前,必须先了解面向对象的3个核心概念。



7.2 面向对象的核心概念

在问题空间中,将客观世界的实体称为对象,不同对象之间的相互作用和互相通信构成了完整的客观世界。如何将问题空间的这一思维模型直接映射到程序空间,也就是说,面向对象语言提供什么概念来支持这一思维模型?综观诸多面向对象的程序设计语言,最核心的概念是数据封装(抽象)、继承和多态性。



7.2.1 数据封装

使用传统设计方法的程序员往往有着共同的弱点。用一个较为夸张的例子来说,当人们问及这些程序员时间的时候,他们会详细地解释钟表的概念。这是由于他们使用计算机的经验所致。这里,必须搞清楚的概念是,告诉别人时间和让他人理解时间的概念是两个不同的知识领域。比方说,某人有一块手表,并经常告诉别人现在是几点了,这并不等于说,他知道这块表的内部工作原理。手表是一个对象,只需知道它提供的是目前的时间,无须了解它如何运转。数据封装的概念反映的就是这一简单原理。

抽象是指对于一个系统的简化的描述。对于使用系统的人员,不会去关心该系统的组成和工作的原理;他们所关心的是,该系统具有什么样的功能,如何去使用该系统(既系统提供什么样的接口,让人们使用)。当然,对于该系统的实现人员,需要关心的是该系统的一切情况。

抽象的原则,运用在计算机领域,称为“信息隐蔽”原则;在面向对象的程序设计语言中,使用数据封装机制实现信息隐蔽。

数据封装将一组数据和这组数据有关的操作封装在一起,形成一个能动的实体,称为对象。用户不必知道对象行为的实现细节,只需根据对象提供的外部特性接口访问对象。例如,可能使用一个数组来存储在屏幕上画一个字符所需要的字形信息:

```
int font[num];
```



如何显示、缩小、放大、增加亮度和设置字符颜色呢？在 C 语言中，常用的解决办法是，将数据结构和相关的函数放入一个可编译的源文件中，把数据和函数作为模块看待。这当然是朝正确的方向迈出了一步。但这还不够好，在数据和函数之间没有明确的关系。不用上述提供的相关函数也能直接操纵 font 的数据，这样可能导致一些问题。假如你决定用链表取代数组，而另一位做同一工作的程序员则可能认为，他有更好的方法来访问这些字形数据，于是他编写了一些自己能直接操纵这些数组的函数。可问题是，那里已经没有数组了！在程序联调时，将产生错误。实际上，在数组 font 和操纵它的函数（显示、缩小、放大、旋转等）之间存在着明确的关系。数组 font 以及这些函数的实现细节对使用者并不重要，重要的是这些函数的界面，使用者只需根据这些函数的界面（函数名及其参数）和函数的功能进行访问。那么，在设计、实现、维护和重用程序时就有很大的帮助。

面向对象语言通过建立一个合适的 Font 类，将字形数据（称为数据成员）和函数（称为成员函数）结合在一起，形成一个新的抽象数据类型，称为类类型（Class）。

```
class Font
{
    字形数据成员；
    操作字形数据的成员函数；
};
```

在这样建立的 Font 类中，可以确保字形数据只能由类中的成员函数进行访问和处理。在任何时候，都可以自由地把字形数据从数组变为链表，只需改变成员函数的实现细节即可。由于这些成员函数的界面不改变，系统其他部分的程序（及使用者）就不会由于改动而受到影响。

类的概念将数据和与这个数据有关的操作集合封装在一起，建立了一个定义良好的接口，人们只关心其使用，不关心其实现细节。这反映了抽象数据类型的思想。

类本身还是一组对象共同属性和操作的抽象。类代表了一般性，而该类的每一个对象代表了具体性。

7.2.2 继承

继承是面向对象语言的另一个重要的概念。在客观世界中，存在着整体和部分的关系（a part of）、一般和特殊的关系（is a 或 a kind of）。

继承实现了一般和特殊的关系，解决了软件的重用性和扩充性问题。

例如，昆虫学家对昆虫的分类如图 7-2 所示。

在试图对一些新的昆虫进行分类以前，关心的问题是：它与其他一般的类有多少相似之处？差别有多大？每一个不同的类都由一组描述它的行为和特征组成，最高层（根结点）是最普遍的，问题也最简单：有翅膀还是没翅膀？每一层都比它之前的层更具体。一旦某个特征定义下来，所有在它之下的种类都要包含该特征。所以，一旦确定蜻蜓为昆虫有翅类中的一种，就不必指出蜻蜓是昆虫，有翅膀，因为蜻蜓从它的目中继承了这个特征。

在面向对象语言中，类功能支持这种层次机制。除了根结点外，每个类都有它的超类

(Super Class), 又称父类 (Parent Class) 或基类 (Base Class)。除了叶结点外, 每个类都有它的子类 (Sub Class/Child Class), 又称派生类 (Derived Class)。一个子类可以从它的基类那里继承所有的数据和操作, 并扩充自己的特殊数据和操作。基类抽象出共同特征, 子类表达其差别。有了类的层次结构和继承性, 不同对象的共同性质只需定义一次, 用户就可以充分利用已有的类, 符合软件重用的目标。

再看一个实际例子, 将人当成对象, 人的属性 (如名字、年龄、性别等) 和对人的属性的操作 (取名字、打印一个人的各种信息等) 已经封装在 **Person** 类中了。学生也是对象, 学生是一类特殊的人, 基本的信息和操作已经 **Person** 中定义过了, 为了避免重复定义, 允许用户将学生类 **Student** 说明为 **Person** 类的继承类。学生类 **Student** 和 **Person** 类的继承关系如图 7-3 所示。这样, **Student** 类就可以不必再考虑对于人的基本属性的定义和操作了, 继承机制允许 **Student** 类从 **Person** 类中继承人的所有数据和操作, **Student** 类只需要考虑学生特殊属性的定义和对学生的属性的操作即可。

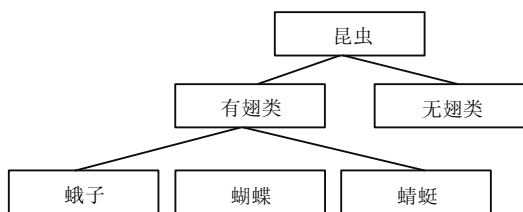


图 7-2 昆虫分类图示

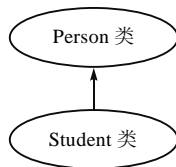


图 7-3 Student 类和 Person 类的继承关系

7.2.3 多态性

面向对象另外一个核心概念是多态性。所谓多态, 是指一个名字 (或符号) 具有多种含义。

在面向对象的程序设计语言中, 多态是通过重载 (或称为超载 **Overload**) 来实现的。

面向对象的程序设计语言中允许两种方式的重载: 函数重载和运算符重载。

1. 函数重载

函数的名字有两个作用:

- ① 代表了该函数的函数体 (那段代码);
- ② 代表了该函数的功能。

在传统的语言中 (如 C 语言), 不允许函数有同名的情况, 考虑的是函数名的第一个作用; 而在面向对象的程序设计语言中, 如果发现多个函数的功能基本是一致的, 尽管它们确实是不同的函数, 但允许它们具有相同的函数名字, 即存在同名函数。

下面考察多态性问题的一个类比问题。当一位汽车司机为避免撞车而紧急刹车时, 他关心的是快速刹车 (效果), 而不关心刹车是鼓式刹车还是盘式刹车 (实现方法的细节)。这里, 刹车的使用与刹车的结构是分离的概念。可能有多种结构的刹车, 它们的使用方法是相同的。相同的使用方法 (相同界面) 对应于不同种类的刹车结构 (多种实现), 这反映了多态性的思想。



与此类似,用户在使用函数编程时,关心的是该函数的功能及其使用界面,并不需要了解该函数的使用界面与函数的哪一种实现方法相匹配 (**Binding**)。也就是说,在设计这一级上,软件人员只关心“施加在对象上的动作是什么”,而不必牵涉“如何实现这个动作”以及“实现这个动作有多少种方法”的细节。例如:

```
int Value(){...}  
void Value (int anInt){...}
```

这两个函数都具有相同的函数名 **Value**,但其参数不同,它们的函数体也可以完全不同,编译能根据其函数参数的不同而自动选择相应的函数体进行匹配。因此,一个函数名代表了多种函数的实现。

对于函数重载,若函数调用(界面)与对应的函数体(函数实现)相匹配是在编译时确定的,则称为早期匹配 (**Early Binding**),或静态联编。若函数调用与对应的函数体的匹配是在运行时动态进行的,则称为晚期匹配 (**Lately Binding**),或动态联编。一般来说,早期匹配执行速度比较快,晚期匹配提供灵活性和高度的问题抽象。

面向对象语言中的虚特性的函数 (**Virtual Function**) 提供了晚期匹配带来的良好特征。函数重载强调的是函数名相同,函数参数和函数体不相同(编译能根据参数的差别进行识别和匹配)。虚特性的函数则强调单界面多实现版本的方法,亦即函数名、返回类型、函数参数的类型、顺序、个数完全相同,但函数体可以完全不同,这在编译阶段是无法识别的,只能由系统在运行时刻动态地寻找所需的函数体进行匹配。

在面向对象的语言中,通过继承关系,基类及其派生类之间构成一个树结构(多重继承为图结构),树中的每个类(基类或派生类)都可以说明一个具有虚特性的函数,称为虚函数。那么在这个类及其派生类中都可以定义这个虚函数的不同实现,但要求这些不同实现必须遵守相同的函数界面,否则虚特性会丢失。使用时,系统能在运行时刻动态地寻找所需的实现版本,详见第 10 章。

面向对象语言中的继承机制和虚函数使得软件可扩充性更为自然。派生类可以继承基类拥有的所有特性,然后加进派生类所需要的特殊的東西,使派生类对象以相似的方式工作并具有新的功能。派生类定义的类型及其虚函数版本是有规则的功能等级的真正扩充。由于这是语言设计的一部分,不是编程时才有的想法,所以很容易实现软件的可扩充性。

2. 运算符重载

C 语言中,有许多系统预定义的运算符例,如“+”,它可以用于 **int** 类型数据,也可用于 **float** 类型数据。虽然使用相同的运算符,但生成的目标代码不相同,因为整数和浮点数在内存中的表示是不同的。这时,“+”运算符具有两种不同的解释(实现代码),也就是说,像“+”这样的运算符在 C 语言中已经被重载。但 C 语言仅支持少量有限的运算符重载。面向对象语言扩充了这个功能,允许已存在的预定义运算符由用户在不同的上下文中做出不同的解释,即:如果类类型的对象使用运算符,则使用的是运算符新的含义;而其他类型的数据使用运算符,使用的是运算符原来的含义。

在原来预定义的运算符含义的基础上,再定义对于某个用户定义类型的对象进行操作的新含义。这就是运算符重载。例如,“+”经过重载后, **str1+str2** 就可以表示两个字符串的合并。



7.3 按对象方式思维

本节介绍一种面向对象设计方法，称为 **CRC** 方法。

到目前为止，已经建立了这样的思想基础：面向对象是一种仿真技术，面向对象设计产生一种与现实具有自然关系的软件系统。那么，怎样用面向对象方法进行设计？怎么按对象方式进行思维？

如果把软件设计理解成一个可操作模型，面向对象设计就是一种最自然的途径。模型化的外部世界都是由对象（传感器、设备、飞机、雇员、工资单、税收等）组成的，软件对象只不过反映了这些外部对象。因此，面向对象设计人员通常不花费时间在学术上讨论寻找对象的方法，而是围绕这些模型化的物理世界的对象来组织模型。

面向过程的程序设计方法鼓励全局地看问题，典型的如结构程序设计，它强调系统功能的逐步细化。使用这种方法开发的程序，其中每个子程序都知道自己来自何处，将要去哪里。而在对象程序设计中，每个对象严格地是局部的。每个对象管理自己的实现，并且不干涉互相的事。用于人，这是一种非常自私的哲学：“各人自扫门前雪，不管他人瓦上霜”。用于模块，这是得到分散式结构的一个必要的需求：一个操作实现的变化只影响包含该操作的对象，一个新类型变量的增加在大多数情况下将使其他对象完全不受影响。

一个旋转图形的例程（**Routine**）没有必要知道所有的图形类是什么。正是这种具有局部性质的对象共同工作的集体效应完成了计算工作。因此，学习面向对象设计，需要人们将思维方式从使用完整的知识全局地看问题，转变到仅根据局部图像寻找对象，并将对象进行组合的方法上来。这就要求集中精力判断哪一个对象应做哪一部分计算，仅当对象做正确的工作时，才能将注意力移到其他问题上，诸如对象怎样被描述，它们怎样被继承等。

设计对象需要做许多小决策，决定怎样将计算的部分分配给不同的对象，称为分散责任（**Distribution Responsibility**），这是设计决策的基础，其余都是次要的。**CRC** 方法帮助设计者分散责任，直到设计的最后阶段才考虑问题的总体。

面向对象的 **CRC** 方法从以下三个方面来表述对象。

（1）Class Name（对象取名）

给对象取名，实际上就是对需要处理的问题空间中包含的不同性质的数据进行分类。在讨论设计模型、实现模型和用户模型时都要使用这些对象（所以称为面向对象的）。在设计早期仔细地选择对象的名字，用直观的语言表达如何组织系统，为以后设计的正确性奠定基础。

（2）Responsibilities（责任）

每个对象完成设计中的一个小目标的功能，它实现了整个系统状态的某一部分。这些责任可以用带活跃动词的短语来描述。就像给对象取名一样，仔细选择描述责任的词也是很有价值的设计活动。

（3）Collaborators（合作者）

每个对象可能会依赖其他对象来完成其责任。这个对象所依赖的对象集合称为它的合作者。重要的合作者能帮助设计者弄清楚怎样将责任在系统中分散。

CRC 设计方法将继承的决策推迟到设计后期甚至实现阶段。初学者如果首先过分地关



注继承，那么他将花去太多的时间来得到比较完善的类分层，忽略了用户的需求和实现过程的实际需要。**CRC** 方法将继承这个有价值的资源放入实现者的手中，实现者能用之来减少代码的行数。继承还有其他有效的使用，诸如协议规格或对象分类。不管怎样选择使用继承，推迟继承策略直到已将系统状态分解为对象之后，将保证能最好地利用继承。

CRC 方法也忽略了对对象描述的问题，到了某一阶段，描述决策将显得很重要：是嵌入进去还是用指针指向，用栈实现还是用堆分配技术等。有了稳定的责任分布会使这些决策容易，因为你已知道，每个对象所涉及的存储信息是什么。

如何判定“是否已经找到合适的对象”，没有一定的准则，这里提出几条判定方法，仅供参考。

较好的 **CRC** 对象应该具有如下特性。

① 从实际比喻中取一个一致性的名字。好的名字常会使设计更清晰。例如，不要将一种画称为“**Paint**”，而另一种画称为“**ColorValue**”，要保持一致性的名字，不然会导致混乱。

② 具有简明扼要的责任。随着对象设计的成熟，不应该用一个对象保存数据，而用另一个对象来处理这些数据。对象应具有责任。

③ 具有较小的但实用的一组合作者。一个对象，如果能同所有的对象打交道，则认为该对象是全局性的象征。属于过程式的思维，应该力图避免这一现象。分解这样的对象，或者把它们的责任分散到已存在的其他对象上，或者创建几个新对象取代之。

④ 责任不要太多。如果一个对象承担的责任太多，寻找更简明的方法来表述对象的责任，例如，创建其他对象来分担一些责任。

较差的 **CRC** 对象具有如下特征。

① 没有任何责任。设计结束时，不承担任何计算负担的对象应该删除。

② 用 **Manager** 命名。像 **Manager**、**Object** 或 **Thing** 这样的词作为对象的名字，没有表达出任何语义信息。应消除这种“噪声”字，寻找更好的、更能表达对象正在做什么的词为对象命名。例如，应使用 **ProcessScheduler**，而不用 **ProcessManager**；应该用 **Figure**，而不用 **DrawingObject**。

③ 使用 **has**、**hold** 或 **uses** 来表达责任。这些词仅提供描述而不提供行为。表达责任的词应尽可能解释为什么其他对象还需要维持。例如，在一个 **Bitmap** 对象中，可能会列出这样的责任“**Holds width and height**”，也可以将其重写为“**Shapes bits into a rectangle**”。后者表达的语义信息显然多一些。

④ 具有庞大的、高度连通的分支。如果一个对象仅关心有限的计算，那么它并不需要与大量的对象通信。良好设计的对象组应该比较小，且它们之间的通信应有限。

设计对象并不难，但在设计对象的同时又要弄清所有的外围关系，这就很难了。**CRC** 方法主要集中于对象怎么称呼，对象能做什么，它能帮助你获得掌握面向对象设计的经验。**CRC** 设计能形成更详细设计的基础，当前的许多设计方法都是基于 **CRC** 的或明显受 **CRC** 方法的影响。

学会在对象之间分散责任是迈向“**thinking like an object**”的第一步。



7.4 面向对象的思想和方法

人们从不同角度看待面向对象,认为它是一种新的系统设计方法,一种新的程序设计方法,一种新的软件构造方法,对象是一种抽象数据类型等。本节试图从面向对象与认知方法学,面向对象与软件 IC,面向对象与结构程序设计方法,对象与抽象数据类型等多个侧面,介绍人们对面向对象的思想和方法的认识。

7.4.1 面向对象是一种认知方法学

从人们的认识过程来看,主要有以下两种方法。

一种是从一般到特殊的演绎方法。以学校为例,最初人们看到的是“学校”这样一个词。在对其进行分类的过程中就可以不断理解这个词的含义:进一步知道学校有大学、中专、中学和小学之分;再进一步又知道大学分综合性大学,理、工、农、医、文科大学等;每一科分为不同专业,专业分为不同方向;……;在分类到一定程度后,用面向对象的方法,就是对已经分好的各类对象进行状态描述和功能定义,以明确这一类对象所能完成的工作,其实这也是一种分类。最后一步就是怎样让这一类对象运转起来,也就是使各类对象建立联系,使用继承和类比方法进行状态转换,完成它们应有的功能。对于学校这一系统,就是建立各教学保障单位与各教学单位的联系,教师与学生的联系,从而使整个教学系统正常运转起来。

另一种是从特殊到一般的归纳方法。今天看到一条黄狗,它是一个对象,明天又看到一条白狗,它也是一个对象,这两个对象除了颜色不一样外,其他有关狗的特征完全一样。这样,便可以构造一个类——“狗”,其中描述了狗的所有共同特征,比如:会叫,具有犬齿,嗅觉灵敏,具有颜色,忠实等。而黄狗和白狗都是这个“狗”类的实例。因此,面向对象很适合这种认知方式的组合。

若定义类时先考虑这一组对象的共性,例如,定义圆类时,所有的圆都有圆心和半径,又如,在考虑继承时,先定义基类,再定义派生类,这些都是从一般到特殊的演绎手段。若定义类时先考虑这一组对象每一个具体对象的属性,例如,定义圆类时,每一个圆都有自己的圆心和半径,又如,在考虑继承时,先定义派生类,再定义基类,这些提供了从特殊到一般的归纳形式。

面向对象提供了从一般到特殊的演绎手段,又提供了从特殊到一般的归纳形式,从而说明它是一种很好的认知方法。这种分类、归纳的方法在面向对象设计技术中是很重要的。

7.4.2 面向对象与软件IC

1. “软件IC”的概念

在软件工程中要提高软件生产率,就应当走类似硬件的道路,应该有软件的“集成电路”(简称软件 IC)。因此,必须寻找一种能够比较容易地将正确成熟的软件单元应用于新的软件系统中的技术。最好有一种集成机制,可以将已经成熟的软件单元制成一个相对独立的实



体,使它们不加改动或很少改动就可以应用于新的软件系统中。只有这样,软件生产才可望减少重复劳动,提高生产率。“软件 IC”的概念就是基于这一思想的一种软件集成机制。“软件 IC”是一种可重用模块。

2. 软件可重用性的问题

避开许多非技术障碍,来考虑软件开发中的重复性质。编程人员一次又一次地重复编制一些基本模式:分类、搜索、读写、分配、同步、比较……但这些事情并不是每次都一模一样的,有如此之多的细节需要变化。因此,软件工程师大多这样做:不断地在同一主题上,精心雕琢一个个新的变种。

针对软件重复的这种性质,如何寻找合适的可重用模块,达到以不变应万变的能力?下面分析构成可重用模块碰到的问题。

例如,一个表的搜索模块,有一些共同的东西:开始在表中的某一位置查找,判断当前位置是不是要查找的,如果不是,移到下一位置。当找到需要的元素或者表查完了也找不到时,结束查找。

但是,细节会有如下的一些变化。

① 类型的变化:元素可能有多种类型,整型元素、一个雇员的记录等。

② 数据结构和算法的变化:表可能是顺序表、数组、二叉搜索树、各种结构的文件等,因此与之相关的搜索算法也随之变化。

③ 相关的程序:为了要知道如何搜索表,搜索程序本身是不够的,它必须知道表是如何创建的,元素如何插入、删除等。搜索程序必须与这些相关的程序相联系。

为了隐蔽上述的这些变化,使得搜索程序在整个软件生存期间,任何表示的变化对用户都是不可见的,亦即用户在执行期间,不应受变化的影响,因此需要表示的独立性。

④ 表示的独立性:真正灵活的模块结构,用户应该仅仅关心操作是怎样使用的,而无须知道它是如何实现的。

例如,对于用户程序,应该含有如下调用:

```
present = search(x, t);
```

在表 t 中,搜索元素 x 。用户调用时并不关心 t 是顺序表,还是二叉树。当运行时,系统应能自动地使这一操作界面 ($\text{search}(x, t)$) 与适当的搜索算法 (search 的实现) 相匹配。上述函数调用是相当高级的,它使用户摆脱了许多实现的细节。但从系统实现的角度,它需要搜索算法的多种实现。这里的问题是,实现人员如何利用这些实现中的共同性,而没有过多的重复。

⑤ 各组实现间的共同性:例如,顺序表的实现。不论它们实现成顺序数组、链表,还是顺序排序文件,算法实质上是一样的,仅有一些差异。应当寻找一种方法,按增量开发的方式来工作。在一般级上给出顺序搜索实现的总模式和性质(例如,在初始位置上启动,前进到下一位置进行检测,直至找到或表查完),然后对每一特殊实现填写细节(例如,前进到下一位置等的实现细节)。类属和继承的概念有助于程序员开发这种共同性。

由以上分析可见,程序设计是一种高度重复的活动,涉及许多共同模式的频繁使用,然而,如何使用和组合这些模式,却有相当大的变化。因此,构造可重用模块需要解决许多技术上的问题,需要程序设计语言的支持。

3. “软件IC”的特征

作为软件的一种集成机制,“软件 IC”必须具有以下特征。

① 模块性好,独立性强。一个“软件 IC”应是可以独立存在的实体,它应当不受或少受外界的影响,以便能较为自由地为各个不同的软件系统所用。

② 可靠性好。因为“软件 IC”是用于构造新的软件系统的基本单位,因此必须保证它具有很好的可靠性。

③ 连接简单,使用方便。为了方便并安全地建造新的软件系统,必须要求“软件 IC”之间连接简单。

④ 封装功能。作为集成机制,“软件 IC”必须将集成的功能实现封装起来,使用户不必搞清楚它的内部细节,加快软件工程的进度。

⑤ 内部功能的高效实现。“软件 IC”是可重用的,其内部代码实现的高效与否直接影响到软件系统的性能。

⑥ 清晰、简明的说明。“软件 IC”是作为产品交给用户的,它应当具有类似于硬件集成插件的功能及各项指标的说明,以使用户可以根据需要选择合适的“软件 IC”。

4. 类概念支持“软件IC”技术

在面向对象语言中,类充当了系统构造的单元,这是因为它具有以下特点。

类是一类对象的统一模板,它具有很强的模块性。类的功能代码实现只通过外部接口与外界联系,具有很强的独立性。

类的可靠性表现在一个类的出错不会传播到其他类中,系统的某个类如果出错,可以将其删除,作为降级的系统使用,而不会使整个系统瘫痪。

类是对象一级的抽象,它将一类对象的数据结构和功能实现封装起来,使得外界不必清楚其内部实现,只要从它的说明中了解其基本功能就可以使用它。

类之间的连接只有继承性描述和一组外协消息,因而清晰、明了。

另外,类支持代码共享,类中的代码均是可再入的,这也支持了可重用性。因此类作为“软件 IC”来组织软件系统从概念上是非常合适的。

要将类改造为更实用的“软件 IC”,还需要许多理论上的支持(类插件语义的形式化研究)和大量的实践,还需克服面向对象技术可能导致产生较慢或较大的程序,或者既大又慢的程序的缺点。

目前,采用面向对象的封装技术生产类似软件 IC 的工作正在进行大量的实践。例如,不同的硬件和软件如何在网络上进行方便的连接,可以采用面向对象的封装技术,形成公共的界面,只要服务程序遵循共同的网络标准,填充公共的 API 统一界面,应用程序就能在不同的网络环境上运行。甚至有的程序员在追求这样的面向对象程序风格:在一个应用模型范围内,当应用对象改变了,程序只需修改数据结构和其初始化就能适应应用对象的变化。在通用表格处理方面已开始出现一些这样的产品。

毋庸置疑,面向对象方法实际上是当前软件开发中最接近问题的一种方法,它对传统方法有很大改进。面向对象的研究还不成熟,不能指望在许多领域马上有类似上述“软件 IC”的产品出售,但在一个开发小组或一个开发项目中,采用面向对象的思想开发和组织软件系



统是很有必要的。

7.4.3 面向对象方法与结构化程序设计方法

结构化程序设计强调了功能抽象和模块性，它将解决问题的过程看做是一个处理过程；面向对象程序设计则综合了功能抽象和数据抽象，它将解决问题的过程看做是一个分类演绎的过程。下面对两种方法进行简单的比较。

1. 模块与对象

在结构化程序设计中，模块是对功能的抽象，每个模块都是一个处理单位，它有输入和输出。而对象则包括数据、操作的整体，是对数据和功能的抽象和统一。可以说，对象包含了模块。

2. 过程调用与消息传递

在结构化程序设计中，过程为一独立实体，显式地被它的使用者所见，而且，对于相同的输入参数，每一次过程调用，其输出结果是相同的。

在面向对象的程序设计中，方法（过程或操作）是隶属于对象的，它不是独立存在的实体，而是对象的功能体现。从对象实现机制看，对象是一个自动机，其中私有状态表示了对象的状态，该状态只能由对象的操作改变它。**Smalltalk** 语言把操作称为方法（**Method**），**C++** 语言用成员函数实现操作。每当需要改变状态的时候，只能由其他对象向该对象发送消息（在 **C++** 中，借助于成员函数调用来实现消息发送）。对象响应消息后，按照消息模式找出匹配的方法，并执行该方法（**C++** 语言是执行函数调用）。应该注意，发送消息和过程调用的意义是不同的，发送消息只是触发自动机，同样的输入参数，可能因为自动机状态不同，其输出结果也不同。因此，同一消息的多次发送可能产生不同的结果。

3. 类型与类

类型与类都是对数据和操作的抽象，即定义了一组具有共同特征的数据和可以操纵这些数据的一组操作，但类所定义的数据集（包括数据和操作）比常规语言的类型所定义的数据集要复杂得多。

例如，在生成实例时，类不像常规语言那样做个声明就可以了。在 **C** 语言中，语句 `int num;` 将 `num` 声明为一个整型实例变量。而类需要事先规定一个生成实例的操作，例如，在 **C++** 语言中的构造函数。当实例变量在程序中不再使用时，类需要使用析构函数（在 **C++** 语言中）或者自动回收机制（如 **Smalltalk** 语言）来回收无用的单元。而且，类引入了继承机制，实现了可扩充性。

4. 模块的可重用性

结构化程序设计方法的核心是逐步细化。这种自顶向下的方法是通过不断在程序的控制结构中增加细节来开发程序的。它生产的模块往往为了满足特定需要，可重用性较差。

面向对象程序设计以数据结构为中心开发模块，同时一体化地考虑操作的功能，抓住了程序设计中最不易变的部分——数据，因此对象常具有良好的可重用性。

7.4.4 对象是抽象数据类型的实现

类型是程序设计语言中一个最基本的概念。数据、函数和过程都有自己的类型，这是构造程序的基础。

人们在生活和工作中，由于不同的目的，常把客观事物加以区别，进行组织，按照事物的特性、行为或用途，对事物进行分类，便产生了类型。

因此，类型化是指以共同的特性和统一的行为把一个可计算的非类型化的世界分解成多个子集，在同一子集中的每个成分有相同的特性和行为。这些子集是可以区别的，构成了不同的类型。

在程序设计语言中，数据类型就是按照以上的思路形成的一个重要概念。一个数据类型可以定义为：

- ◎ 一个值集；

- ◎ 一个作用于值集的操作集。

例如，整数类型，它所包含的值集是：

..., -3, -2, -1, 0, 1, 2, 3, ...

操作集则是：加、减、乘、除、取绝对值等。

抽象是一个系统的简化描述，它强调了系统的某些特性而忽略了其他特性。对于用户来说，所关心的是程序能做什么，而不是它的实现细节。而抽象恰好可以用来对用户所关心的重要信息予以强调，而忽略不重要的信息。整个程序设计语言的发展过程就是抽象层次不断提高的过程。

为了完全、精确、无二义地描述数据类型，而不愿涉及其物理表示和实现的细节，人们采用抽象数据类型的描述技术。

抽象数据类型在程序中是一个封闭的单位。它包含的信息有：

- ◎ 外部可见的数据；

- ◎ 外部不可见的的数据；

- ◎ 施于数据上的各操作的界面（或称为标准接口）；

- ◎ 所有操作的实现细节。

于是，将数据类型抽象为提供给外部世界的数据和操作集。使用了抽象数据类型描述，不再关心数据结构是什么，而仅仅关心它具有什么功能。这与信息隐藏原则很符合。

总之，抽象数据类型从外部观点来描述数据类型：仅关心各种有用的操作和这些操作的性质。可见，抽象数据类型描述的信息与对象包含的信息是一致的，对象是抽象数据类型的实现。

现在可以引用 **Bertrand Meyer** 的一个关于面向对象程序设计技术性的定义：面向对象程序设计是这样一种软件系统构造方法，它把软件系统构造成“各抽象数据类型实现的结构化集合”。这里“结构化”一词反映出各类之间存在着重要的关系，类与类之间可建立层次结构以形成继承关系，类与类之间通过消息传递方式形成“委托—服务”关系等。“集合”表示了各个类的设计和系统构造方法。每个类应设计成独立自治的，尽可能与它们所属的系统



无关，具有良好的可重用性。系统构造可看成是各个类的自底向上的组装。



7.5 面向对象的程序设计语言

面向对象程序设计语言是支持面向对象设计的语言工具。随着人们对面向对象技术的兴趣日益浓厚，面向对象语言的研究也非常活跃。面向对象的概念渗透软件的各个领域。例如，面向对象数据库、面向对象图形处理、面向对象操作系统、面向对象快速原型技术、面向对象分析技术、神经网络中的面向对象技术等。面向对象技术被誉为 20 世纪 90 年代软件的核心技术之一。

将面向对象的概念正式作为语言成分的是 Simula 67，现称为 Simula 语言。Simula 67 语言是 Ole Dahl 和 Krisrten Nygaard 等人于 1967 年设计的。Simula 语言是在 ALGOL 60 语言的基础上发展起来的，引入了类、对象、继承和共行子程序等概念。当今的面向对象程序设计的基本思想来源于 Simula 67 语言，但是现在 Simula 67 语言本身并不用于面向对象的程序设计。

1. Smalltalk 语言

20 世纪 80 年代以后，Smalltalk 语言的问世标志着面向对象语言研究的开始。Alan Kan 在 Xerox Palo Alto 研究中心 (Xerox PARC) 领导一个研究小组设计并实现了 Smalltalk 语言。Smalltalk 语言除了吸取 Simula 语言基本的面向对象的概念外，还受到 Lisp 语言的很大影响：无类型设施、名字的动态结合以及运行时刻类型的动态检查。特别是 Smalltalk 为了概念上的一致，把所有的系统成分都看做对象，并由此引入元类的概念，想用尽可能少的概念表达出各种要求。

Smalltalk 语言具有一个非常好的开发环境。运行 Smalltalk 程序时，会发现自己处于一个完全不同的环境里。在这里，看不到编译、链接甚至 Dedicated Editor。取而代之的是 Browsers, Workspaces, Inspectors, Walkbacks 和在 GUI 下的 Transcript Windows。这个完善的环境帮助用户管理源代码和说明。Smalltalk 的类和方法不仅仅是类库，而且是构成 Smalltalk 环境的一部分，这里的每一件事都同系统的其他部分连在一起。Smalltalk 的整个环境都是面向对象的，至今 Smalltalk 仍被认为是最纯的面向对象语言。Simula 和 Smalltalk 创建了当今出现在面向对象语言中的主要概念。

面向对象的思想是慢慢渗入软件工业界的，这种缓慢的迁移归于大量因素。尽管 Simula 67 和 Smalltalk 在大学圈子里是著名的，但 20 世纪 80 年代初期在软件工业界默默无闻。Smalltalk 的早期研讨主要出现在 Xerox PARC 的出版物上。另外，对计算平台的特殊要求也使得这些面向对象的早期语言的使用很难吸引公司和商用软件开发人员。许多软件开发人员第一次接触 Smalltalk 时，认为它不过是一个窗口系统，很难体会到这是一种新的程序设计风格。

2. C++ 语言

20 世纪 80 年代，C 语言已非常受欢迎，无论微机还是其他结构的计算机和环境都配备了 C 语言。20 世纪 80 年代早期，AT & T 贝尔实验室的 Bjarne Stroustrup 及其研究小组将 C 语言扩展为 C++ 语言——一种支持面向对象概念的 C 语言。当时开发 C++ 语言是为了支持

编写一些复杂事件驱动的仿真程序，以解决用 Simula 67 语言带来的效率方面的问题。C++ 语言中的一些灵感来自 Simula 67 和 ALGOL 68。C++ 语言自 1983 年发表以来，引起了很大反响。AT & T 允诺 C++ 语言与 C 语言保持完全的兼容性，AT&T 公司和大量的软件公司开发了 C++ 语言的几个商用编译和环境软件，从提高其运行效率，使得面向对象程序设计语言在软件工业界受到了广泛的关注，在商业上取得了巨大的成功。因为学习 C++ 语言，程序员能以一种熟悉的、普通的语法学习面向对象的风格，没有必要投入过多精力去学习一种新的、完全不同的计算机语言和环境。

当今个人计算机的硬件配置已经能够满足高性能工作站的基本需要，高质量的图形显示器、工具齐全的开发环境使得阻碍面向对象技术广泛使用的性能价格比障碍大大减小，这加速了面向对象风格的使用。由于缺乏更好的开发手段，软件工业已变得停滞不前，软件工业期盼更好的软件开发方法，面向对象提供了更好的处理软件复杂性的方法。使用 C 语言等开发工具来开发应用的程序员常常感到缺乏抽象层次，面向对象风格则提供了程序设计不同的抽象层次——从对象到类、类库直至整个应用框架 (Frameworks)。面向对象风格能使商用软件开发加速产品开发，方便地扩充代码，并提供在市场上有竞争力的软件产品。这就是 C++ 语言越来越受欢迎的原因之一。C++ 语言目前已成为在实用上最有前途的面向对象语言。

3. Java语言

Java 语言是一种通用、并发、基于类的面向对象的程序设计语言。Java 语言的名字来源于印度尼西亚的一个岛名“爪哇”。

Java 语言诞生于 1991 年，当时 Sun Microsystems 公司的 James Gosling 领导的 Green 小组试图开发一种面向消费类数字设备的语言。1992 年夏，他们实现了第一个版本，但他们的工作太超前了，未得到业界的关注和接受。

1993 年，出现了 Mosaic 网络浏览器，这对 Internet 从学术领域走向商业领域起到了推动作用。Green 小组立刻认识到他们开发的语言可以用来提高浏览器的性能。因为浏览器需要执行若干协议，网络到用户的数据传输速率又很有限，所以当网络用户等待信息显示时，他们的计算机常常处于空闲状态。Sun Microsystems 公司认识到他们开发的语言在网络上很有价值。1994 年，Sun Microsystems 公司发布了包含 Java 虚拟机的 Hotjava 浏览器。1995 年 5 月 23 日，Netscape Communications 公司的创始人之一 Marc Andressen 宣布 Netscape 将在其 Netscape 浏览器集成 Java 虚拟机，而此时的 Netscape 已占有 70% 的浏览器市场。从那以后，Java 语言的应用迅速增加，虽然设计 Java 语言的目的是开发网络浏览器的小应用程序，但是作为一种通用的程序设计语言，Java 语言已被广泛接受，并且有可能代替 C 语言和 C++ 语言成为业界首选的编程语言。

Java 语言类似于 C 语言和 C++ 语言。从历史上看，它们之间有一定的关系。20 世纪 70 年代，C 语言作为一种开发操作系统的语言出现，因此，C 语言的设计者主要是想开发一种允许访问计算机底层结构的语言。Stroustrup 在开发 C++ 时，从 SIMULA 语言中引入包的概念，从 Smalltalk 语言引入继承的概念，但基本的 C 语言并未修改。所以，C++ 语言沿袭了 C 语言所具有的便于开发系统程序的特点。当 Sun Microsystems 公司开发 Java 语言时，他们保留了 C++ 语言的语法、类和继承等基本概念，删除了一些不好的特征。因此，Java 语言是一个比 C++ 语言更简单的语言。作为一种有用的语言，其语法和语义比 C++ 语言更合理。



可以说, Java 语言是在 C 语言和 C++语言基础上开发出来的语言, 但又吸收了其他语言的一些有益的成分。类的概念来自于 C++语言和 Smalltalk 语言, 但 Java 语言只限于单实现继承。接口的概念来自于 Objective-C 语言, Java 语言提供多接口。包的概念来自于 Modula 语言, 在 Java 语言中增加了层次性名字空间和逻辑开发单元。并发的概念来自于 Mesa 语言, Java 语言内置了多线程支持。异常处理的概念来自于 Modula 3 语言, 在 Java 方法中增加了抛出异常的说明。动态链接与自动内存回收的概念来自于 Lisp 语言, Java 的类可以在需要时装入内存, 不需要时将其释放。

Java 语言与其他语言的不同之处在于, 它为减少与具体实现的相关性方面, 做出了特别的努力。Java 语言允许程序员只编写一次代码, 就可以使其代码在 Internet 上的任何地方毫无阻碍地运行。Java 语言的设计目标是, 只要给定足够的内存空间和时间, 一个确定的程序无论在任何机器上, 以何种实现方式运行, 总能得到相同的运算结果。几年间, Java 语言已从“诞生”走向“成熟”, 并得到了广泛应用。同时, 它还在不断地发展, 并不断推出新的版本。这种改进和完善是建立在同原有版本完全兼容的基础之上的。Java 语言的特性很多, 有些特性将在数据类型和控制机制中讨论, 这里讨论它的一部分特性。

(1) 面向对象

过程式语言对问题求解的描述, 是通过对计算机执行的一系列步骤的描述来实现的, 而面向对象语言则用问题空间中的元素与对象来描述问题。这样, 一个好的设计就可以得到可复用、可扩充和可维护的组件。这些组件可以灵活地适应处理环境的改变, 因为对象之间的主要工作就是来回发送消息。

Java 语言基于类, 类的实例就是对象。面向对象是一种程序设计方法, 这些方法与程序设计语言一样, 要经过“诞生”、“成熟”和“消亡”的历程。面向对象方法正处于“成熟”阶段。但是, 现在也有人对它提出批评, 认为它将被构件方法所代替。

(2) 解释性

Java 源程序编译成平台中立的字节码, 这些字节码可以传输到任何具有 Java 语言运行环境的平台, 其中包括 Java 虚拟机, 从而在运行时不需要重新编译和重新链接。Java 解释器通常在网上运行, 网上信息传输相对来说比较慢, 在等待传输的空闲时间就可以解释执行 Java 字节码。另外, 开发商还提供了一种即时 (Just-in Time) 编译器, 对那些要求快速运行的程序进行即时编译, 并充分考虑了优化, 从而达到了与 C++语言同样的执行速度。

(3) 简单性

Java 语言的风格类似于 C 语言和 C++语言, 因而, 熟悉 C 语言和 C++语言的程序员可以很快掌握 Java 语言的编程技术。Java 语言提供了丰富的类库, 使编程变得比较简单。Java 语言抛弃了指针, 明确了类型转换的语义规则, 降低了程序出错的可能性, 使程序逻辑更清晰。

(4) 高效性

Java 语言的多线程并发执行, 节省了 CPU 的空闲时间。高效字节码与机器代码的执行效率相差无几。

(5) 动态性

Java 语言实现动态内存管理, 能及时回收无用存储单元。

(6) 分布性

Java 语言的动态特性使它在分布式环境中,尤其是在 Internet 环境中,提供方便的动态内容支持。

(7) 健壮性

健壮性主要反映程序的可靠性。Java 语言为此提供了大量的支持。Java 语言是强类型语言,制定了严格的编译时类型检查规范。Java 语言没有指针,减少了许多隐藏错误。Java 语言动态自动回收无用存储单元(释放无用单元),使程序员无须进行内存管理。Java 语言鼓励用接口而不是使用类。接口定义一组行为,而类实现这些行为。传递接口而不传递类,从而可隐藏这些实现细节。若要改变实现细节,只需要新类实现旧接口,其余一切照常工作。

(8) 安全性

Java 语言定义了严密的安全规范,从而保证程序的安全执行。

(9) 可移植性

Java 程序一次编写可到处运行。Java 语言为内部类型的实现定义了严密的规范,不会出现各种不同的“方言”。

(10) 并发性

Java 语言支持多线程,从而支持程序在单机上的并发执行,或者在多机上的并行运行。Java 语言以监控状态模型为基础提供了对同步的支持。

(11) 平台无关性

Java 程序是由 Java 虚拟机来执行的。Sun 公司撰写了“Java 虚拟机规范”,各操作系统供应商提供与平台相关的符合规范的 Java 虚拟机。Java 源程序经编译成字节码后,就可以由各种不同操作系统上的遵守相同规范的 Java 虚拟机来执行,且能获得相同的执行结果。这就是 Java 跨平台的秘密,从而实现了 Java 语言与平台无关性。事实上,Java 语言的跨平台特性是以 Java 虚拟机不能跨平台为代价的。如果我们设想,BASIC 语言有一个“BASIC 虚拟机规范”,各操作系统供应商按此规范提供各自的 BASIC 虚拟机,那么,BASIC 语言也就具有跨平台的特性了。

4. C#语言

C#(C Sharp)语言是微软开发的一种面向对象语言,其目标是既拥有 C++语言的执行效率和运算能力,也具备如 VB 语言一样的易用性。C#语言是基于 C++语言的一种语言,同时包含类似 Java 语言的很多特征。

对于 C/C++语言用户来说,最理想的解决方案无疑是在快速开发的同时又可以调用底层平台的所有功能。他们想要一种与最新的网络标准保持同步,并且能与已有的应用程序良好整合的环境。另外,一些 C/C++语言开发人员还需要在必要的时候进行一些底层的编程。

C#语言是一种最新的、面向对象的编程语言。它使得程序员可以快速地编写各种基于 Microsoft.NET 平台的应用程序。

正是由于 C#语言面向对象的卓越设计,使它成为构建各类组件理想的选择——无论高级的商业对象还是系统级的应用程序。使用简单的 C#语言结构,这些组件可以方便地转化为 XML 网络服务,从而使它们可以由任何语言在任何操作系统上通过 Internet 进行调用,即任何平台的应用程序都可以通过 Internet 调用它。



扩展交互性作为一种自动管理的、类型安全的环境，C#语言适用于大多数企业应用程序。但实际的经验表明，有些应用程序仍然需要一些底层的代码，要么是因为基于性能的考虑，要么是因为要与现有的应用程序接口兼容。这些情况可能会迫使开发者使用 C++ 语言，即使他们本身宁愿使用更高效的开发环境。C#语言采用以下对策来解决这一问题：

- ① 内置对组建对象模型（COM）和基于 Windows 的 API 的支持；
- ② 允许有限制地使用纯指针（Native Pointer）。

在 C#语言中，每个对象都自动生成为一个 COM 对象。开发者不再需要显式地实现 IUnknown 和其他 COM 接口，这些功能都是内置的。类似地，C#语言可以调用现有的 COM 对象，无论它是由什么语言编写的。

C#语言是一种现代的面向对象语言。它使程序员快速便捷地创建基于 Microsoft.NET 平台的解决方案。

C#语言增强了开发者的效率，同时也致力于消除编程中可能导致严重后果的错误。C#语言使 C/C++ 程序员可以快速进行网络开发，同时也保持了开发者所需要的强大性和灵活性。

C#语言是被设计工作在 Microsoft.NET 平台上的，微软的目标是使数据和服务的交换在网页上更容易实现，并且允许开发人员构建更高的程序可移植性。C#语言可以方便地用于 XML 和 SOAP，并可以直接访问程序对象或方法，而不需要添加额外的代码。所以程序可以构建在已存在的代码上，或者多次重复使用。C#语言的目标是更快捷地为市场开发产品和服务且成本开销更低。

微软与 ECMA（国际标准组织）合作，建立了 C#语言的标准。国际标准化组织（ISO）称赞 C#语言可以鼓励其他公司开发属于自己的产品。

C#语言已经被 Apex Software, Bunka Orient, Component Source, DevSoft, FarPoint Technologies, LEAD Technologies, ProtoView 和 Seagate Software 等公司采用。

面向对象方法仅仅是传统程序设计方法的很大改进。面向对象的概念和方法是当前软件开发中最接近问题的一种较好方法，仍具有强大的生命力。目前，面向对象的研究还不成熟，不能指望面向对象的概念能解决软件复杂性的问题，但在一个开发小组或一个项目的开发中，采用面向对象的思想，从可重用性、可扩充性和可兼容性的角度出发，开发和组织软件系统是非常必要的，这也是一种良好的选择。

第 8 章 C++语言实现数据封装

C++语言使用类类型实现数据封装。面向对象的其他核心概念——继承和多态，都是围绕着类类型展开的。



8.1 类的定义

类类型的一般形式为：

```
class ClassName
{
    数据成员声明;
    函数成员声明或定义;
};
```

这里定义了一个新的数据类型：**ClassName**。它既是类的名字，也是该类型的类型名。在{}以内的部分称为类内，{}以外的部分称为类外。

在一个类的内部，可以根据实际需要将成员分成几组，其中，一组是可以在类外进行访问的；一组只能在类内访问，而在类外是不可访问的，或者说，是不可见的。C++提供了实现这个目标的完整机制，就是使用访问控制。

访问控制的实现方法就是将成员按需分散到三种定义段中。这三种定义段分别是 **public**、**private** 和 **protected**。形式化的定义如下：

```
class 类名
{
    [[private:]]          //定义私有段成员
        私有段数据声明;
        私有段函数定义;]

    [protected:          //定义保护段成员
        保护段数据声明;
        保护段函数定义;]

    [public:              //定义公有段成员
        公有段数据声明;
        公有段函数定义;]
};
```

这里，类名是一个标识符，代表类类型的类型名；**private**、**protected** 和 **public** 称为段约束符，其中 **private** 可以省略（即没有被段约束符说明的成员就是私有段成员）。

在 C++语言中，类外部的函数或者其他的类只能通过使用类的公有段成员来访问这个



类。因此，类的公有段成员（公有段数据和公有段成员）提供了类的外部接口，而私有段成员以及所有成员函数的实现细节（函数体部分）则由类封装起来，让类的使用者看不到。类的私有数据只能严格通过成员函数访问，任何类外（除了友元）对私有数据的访问都是非法的。使用私有数据这一语言特性来隐藏由类对象操纵的数据，然后提供一些成员函数来访问这些数据，但隐藏了改变这些数据的能力和实现细节。这样，使得类对数据的描述和类提供给外部世界来处理数据的接口这两件事互相独立，这就给出了面向对象的重要性。一个类的重要性在于它对外部世界的一组对象描述了同一接口，至于怎样去实现这些功能，仅仅是类内部关心的事。各个对象通过该接口完成各项功能。

公有、保护和私有成员的声明顺序可以是任意的。另外，在类里可以有多个公有段、保护段和私有段。

典型的类的定义参见例 8-1。

【例 8-1】日期类定义。

```
class Date
{
private:
    int      day, month, year;
public:
    void      InitDate(int d, int m, int y);
    void      AddYear(int y);
    void      AddMonth(int m);
    void      AddDay(int d);
};
```

通常把具有同样性质和功能的东西所构成的集合叫做类。在 C++ 语言中，也可以把具有相同内部存储结构和相同一组操作的对象看成属于同一类。在指定一个类后，往往把属于这个类的对象称为类的实例。

类只是一种形式化的定义。要使用类提供的功能，必须使用类的实例（类的静态成员例外）。类的实例就是对象。一个类可以定义多个对象，每个对象都包含类中定义的各个数据成员的存储空间，它们共享类中定义的成员函数。

定义一个类对象就像定义一个整型变量一样，例如：

```
Date d;    //定义一个名为 d 的 Date 对象
```

类与对象的关系也类似于整数类型 `int` 与整型变量的关系。类和整数类型 `int` 代表的是一般的概念，对象和整型变量代表的是具体的实例，而每一个实例都要在内存中占据一定的存储空间。

可以将类和对象的关系归纳如下。

- ① 类代表了一组对象的共同性，对象代表了具体的性质。
- ② 类在概念上是一种抽象机制，它抽象了一类对象的存储和操作特性。
- ③ 在系统实现中，类是一种共享机制，它提供了一类对象共享其类的操作实现。
- ④ 类是一种封装机制，它将一组数据和对该组数据的操作封装在一起。

⑤ 类是对象的模型，对象承袭了类中的数据和方法（操作），只是各实例对象的数据初始化状态和各个数据成员的值不同。

例如，圆，所有的圆都应该有圆心和半径两个属性；每一个具体的圆都应该有自己的圆心和半径。

```
class Circle
{
private:
    int    x;           //圆心的 x 坐标
    int    y;           //圆心的 y 坐标
    float  fRadius;     //半径

public:
    void  SetXY(int a,int b);
    void  SetRadius(float r);
    void  Move(int newx,int newy);
    //其他成员
};
```

所有的圆对象都具有自己的圆心和半径，而共享所有对圆的操作。换句话说，就是所有对象共享成员函数的代码。图 8-1 示意了 3 个对象的情况：

```
Circle c1, c2, c3;
```

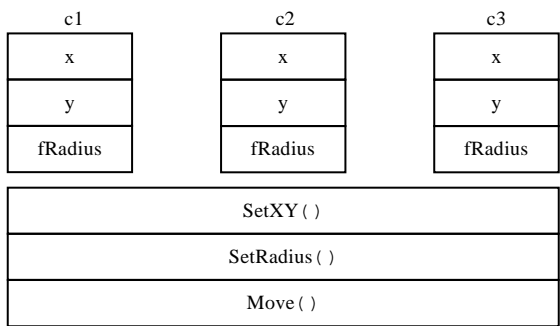


图 8-1 类和对象的关系



8.2 类的成员

一个类类型中可以有二类成员：数据（Data）和操作（Operation）。在 C++语言中，称它们为数据成员（Data Member）和成员函数（Member Function）。在面向对象的术语中，它们分别叫做属性（Attribute/Property）和方法（Method）。

8.2.1 数据成员

类的数据成员往往描述了该类对象所处的状态，因此它被称为属性。所以，在设计一个



类的时候，都会从该类对象中提取那些能表达对象本质的数据作为属性。例如，在设计“学生”这种类时，使用学号、姓名、性别等数据来作为该类的成员。

从理论上讲，类的数据成员的类型可以是任意已经定义的类型，包括编译器内建类型和用户自定义类型。前面的例子都示意了以简单类型的变量作为数据成员的情况，下面的例子说明了在一个类中包含了结构对象的情形。

```
struct Point
{
    double x, y;
};
class Triangle
{
    private:
        Point  v[3];          //三角形的 3 个顶点
        int    nColor;
    public:
        //其他公有成员
        ...
};
```

在一般情况下，一个类不能包含该类类型的对象，因为这样做会使一个类的存在依赖于该类自己。很明显，这是一种递归定义方式，而 C++编译器不支持这样的方式。不过，可以在类中定义指向该类对象的指针。链表中结点的类型定义就是一个很好的例子：

```
class Node
{
    private:
        int      nData;
        Node *   pNext;
    public:
        //其他公有成员
        ...
};
```



8.2.2 成员函数

函数作为成员是面向对象的一种标志。这表明，操作是隶属于对象的。

成员函数与一般的函数一样，都有这样一些特性，如函数名、参数列表和返回类型。不过，有些特殊的成员函数可能没有返回类型。没有返回类型是指函数声明前不能加任何的类型说明，包括 void 在内。这与函数没有返回类型是不同的。

作为成员的函数可以在类类型定义时给出，也可以在类类型定义时只给出成员函数的声明。

1. 在类中定义成员函数

```
class Date //简化版
{
    private:
        int day, month, year;
    public:
        void InitDate(int d, int m, int y)
        {
            day = d;
            month = m;
            year = y;
        }
    ...
};
```

2. 在类中声明成员函数，在类外定义成员函数

在类型定义外再给出函数的定义的格式为：

```
返回类型 类名::函数名(参数列表)
{...}
```

如：

```
class Date //简化版
{
    private:
        int day, month, year;
    public:
        void InitDate(int d, int m, int y);
    ...
};

void Date::InitDate(int d, int m, int y)
{
    day = d;
    month = m;
    year = y;
}
...
```

无论哪种方式，都可以看到，在一个成员函数内部（函数体内），不需要显式指明访问的数据属于哪个对象的，因为成员函数“知道”它正在操作的数据是属于激活该函数的那个对象的。



8.2.3 静态成员

关键字 `static`（静态的）可以用于说明一个类的成员（包括数据成员和成员函数），这样的成员被称为静态成员。

1. 静态数据成员

在一个类中，若在一个数据说明前加上 `static`，则该数据为静态数据。静态数据成员被该类的所有对象所共享。无论建立多少个该类的对象，都只有一个静态数据的存储空间。

静态成员的存储空间必须在类定义外进行分配，具体的语法如下：

类型名 类名::静态数据成员 [= 常量表达式];

其中，常量表达式用于初始化类的静态数据成员。由于静态数据成员的初始化工作不会自动进行（若一个函数内的静态数据没有显示初始化，则自动初始化为 0），因此这个工作是必需的。

图 8-2 形象地说明了类的普通数据成员和静态数据成员的关系。类 `aClass` 包括一个普通数据成员 `anInt` 和一个静态数据成员 `anSint`。假设定义了三个对象 `a`、`b` 和 `c`。可以看出，每个对象都有属于自己的普通数据 `anInt` 的存储空间，共享静态数据 `anSint` 的存储空间。因此，如果每个对象修改了自己的 `anInt`，这不会影响别的对象；而如果一个对象修改了 `anSint` 的值，则这个修改将会对所有的对象有效。

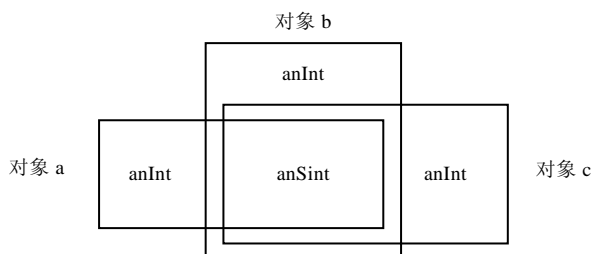


图 8-2 普通数据成员和静态数据成员的关系

静态数据成员属于类，而不属于对象，因为静态成员的存在是不依赖于某个具体的对象的。在这个意义上，在类对象不存在的情况下，也可以访问到类的静态成员。不过，静态数据成员也分为公有和私有的，所以在类外只能访问公有的静态数据成员，且访问方式为：

类名::静态公有数据成员

而在类内可以访问所有的静态数据成员，并且采用直接访问的方式。例 8-2 表示静态数据的说明和初始化。

【例 8-2】

```
#include <iostream>
using namespace std;
class Counter
{
private:
```



```

        static int nCount;
public:
    void SetCount(int num)
    {
        nCount = num;
    }
    void ShowCount()
    {
        cout << nCount << ' ';
    }
};
int Counter::nCount = 0;
void main()
{
    Counter a, b;
    a.ShowCount();
    b.ShowCount();
    a.SetCount(10);
    a.ShowCount();
    b.ShowCount();
}

```

输出为:

```
0 0 10 10
```

要说明一个数据成员为静态的，就是在该数据成员之前加上关键字 `static`。在类说明中，静态数据成员的声明不是定义，因此必须在类说明之外提供定义，以分配存储空间。这里，在类外用下述语句对静态变量 `nCount` 进行定义并初始化为 0：

```
int Counter::nCount = 0;
```

将静态数据 `nCount` 初始化为 0，故第一次对 `ShowCount()` 的调用都显示 0。对象 `a` 把 `nCount` 置为 10，然后 `a` 和 `b` 都用 `ShowCount` 显示 `nCount` 变量值。由于 `a` 和 `b` 共享一个 `nCount`，所以它们都显示 10。

静态数据的典型应用参见例 8-3。

【例 8-3】在仓库管理中，假定现在只关心货物的重量，可以使用静态数据表示总重量。

```

#include <iostream>
using namespace std;
class Goods
{
private:
    float          fWeight;

```



```
static float    fTotalWeight;

public:
    void AddGoods(float weight)
    {
        fWeight = weight;
        fTotalWeight += weight;
    }
    void RemoveGoods()
    {
        fTotalWeight -= fWeight;
    }
    void ShowWeight()
    {
        cout << fWeight << endl;
    }
    void ShowTotalWeight()
    {
        cout << fTotalWeight << endl; //直接访问静态数据
    }
};

float Goods::fTotalWeight = 0.0; //static 成员的定义及初始化
void main()
{
    Goods goods1, goods2, goods3;
    goods1.AddGoods(100);
    goods2.AddGoods(70);
    goods2.ShowWeight();
    goods3.AddGoods(150);
    goods1.ShowTotalWeight();
    goods2.ShowTotalWeight();
    goods3.ShowTotalWeight();
}
```

输出结果为：

```
70
320
320
320
```

在类中定义了成员函数 `ShowTotalWeight()` 来直接访问静态数据 `fTotalWeight`，但由于

fTotalWeight 是所有对象共享的，因此最后 3 条语句都输出同样的结果。

2. 静态成员函数

成员函数也能被说明为静态的。与静态数据成员一样，静态成员函数属于类而不是某个类对象。因此，在类外调用一个公有静态成员函数，不需要指明对象或指向对象的指针。具体格式为：

类名::静态公有成员函数名(参数列表)

而在类内，采用直接调用静态成员函数（包括公有的和私有的）的方式。

一般使用静态成员函数来访问静态数据成员。若在静态成员函数中要访问非静态成员，必须明确指出该静态成员函数在哪个对象上操作，并且必须有限定词（即需要通过对象或指向对象的指针进行），一般需要类类型作为静态成员函数的参数类型（除非是使用全局对象）。静态成员函数的定义和使用见例 8-4。

【例 8-4】

```
#include <iostream>
using namespace std;
class INTEGER
{
private:
    int          member_int;
    static int    static_int;
public:
    static void func(int numb1, int numb2, INTEGER * ptr);    //静态成员函数
    ... //其他成员函数
};
int INTEGER::static_int = 0;
void INTEGER::func(int numb1,int numb2, INTEGER * ptr) //类外实现时，不加 static
{
    //member_int = numb1;        //错误，不知对哪个对象的 member_int 进行
    ptr-> member_int = numb1;    //正确  ptr 仿造了 this 指针的作用
    static_int = numb2;         //正确
}
void main()
{
    INTEGER obj;
    INTEGER::func(1, 2, &obj);    // 正确，但仅对静态成员函数正确
    obj.func(3, 4, &obj);        // 正确
}
```



8.2.4 类外访问成员的方法

有了类对象，可以通过该对象在类外访问该对象的成员。具体的语法格式是：

对象名.公有段的成员函数名(参数列表)

对象名.公有段的数据

指向对象的指针->公有段的成员函数名(参数列表)

指向对象的指针->公有段的数据

例如：

```
Circle cir, *pCir; //定义一个 cir 对象和一个指针变量
```

```
pCir = &cir;
```

```
cir.SetXY(0, 0);
```

```
pCir->SetXY(0, 0);
```

如果要访问类的静态成员，那么可以使用前面的语法。不过更直接的方法是采用下面的语法格式：

类名::公用静态数据成员

类名::公有静态成员函数(参数列表)

例如：

```
INTEGER::func(1, 2, &obj);
```



8.3 C++语言的类

C++语言的类共有 3 种方式：class、struct 和 union。

1. 类class

class 是 C++语言中最典型的抽象数据类型。除了一般的类，我们还可以定义几种特殊的类。

(1) 无名类

若一个类没有类名，则称为无名类（或匿名类）。

```
class //这个类没有名字
```

```
{
```

```
...
```

```
} anObject;
```

无名类只能用做对单个对象的声明。

(2) 空类

```
class empty
```

```
{ };
```

空类没有任何成员，包括数据和函数。空类可以定义多个对象，各个对象具有不同的地址。

2. 结构struct

在 C++ 语言中, 结构是另外一种形式的类。C++ 语言中的结构也能像 `class` 类一样包括数据和成员函数。C++ 语言的结构和类的差别在于, 默认访问控制描述时, 类的成员都是私有的, 而结构的成员都是公有的。除此之外, 类与结构有完全相同的功能。所以结构又称为其全部成员都是公有成员的类。

结构包括的数据成员和成员函数都是公有的。如果要在结构类中定义私有数据, 就需要显式地给出关键字 `private`。

在一般情况下, C++ 语言程序员都用 `class` 来定义对象的抽象形式, 而用 C 语言的方式来使用结构。

3. 联合union

联合是将所有元素都存储在同一位置上的结构。在 C++ 语言中, 联合也是一种类。联合的所有成员只能为公有成员。关键字 `private` 不能用于联合。



8.4 数据封装和信息隐蔽的意义

一个类的定义实现了数据封装。

(1) 类外不可见 (即封装起来的部分)

◎ 私有段数据

◎ 私有段函数原型

◎ 私有段函数和公有段函数的实现

(2) 类外可见 (即一个类向外提供的部分)

◎ 公有段数据

◎ 公有段函数原型

注意: 数据封装是一个相对的概念, 只是对于类外而言的, 而对于类内, 所有成员都是互相可见的。一个类的成员函数的实现, 可以在类内完成, 也可以放在类外实现, 但函数本身仍然是类内的函数。

数据封装将一个数据和与这个数据有关的操作集合封装在一起, 形成一个能动的实体, 称为对象。用户不必知道对象行为的实现细节, 只需根据对象提供的外部特性接口访问对象。

类是 C++ 语言的关键概念, 是一种用户定义的类型; 类是数据隐藏和封装的单位, 它将细节封装起来, 只允许通过公有段的数据和函数被访问, 从而支持了数据抽象的机制。默认时 `class` 类包括的数据成员和成员函数都是私有的。如果要在 `class` 类中定义公有数据, 则需要显式地给出关键字 `public`。

在 C++ 语言中, 私有数据只能严格通过成员函数访问, 任何类外 (除了友元) 对私有数据的访问都是非法的。使用私有数据这一语言特性来隐藏由类对象操纵的数据, 然后提供一些成员函数来访问这些数据, 但隐藏了改变这些数据的能力和实现细节。这样, 使得类对数



据的描述和类提供给外部世界来处理数据的界面这两件事互相独立,这就给出了面向对象的重要性。一个类的重要性在于它对外部世界描述了操作的界面,至于怎样去实现这些操作,仅仅是类内部关心的事。

将对象的描述、对象操作的实现与对象提供的界面分开,使得类的使用者(委托方)对类的实现部分的依赖程度大大减小。在面向对象方法中,一个类的用户唯一需要做的事情是访问类的界面。这样,用户能随意改变对象的描述和对象操作的实现而不必通知任何委托方,不必要求改变他们依赖这个类做的工作。这使得用户扩充类、修复类的能力大大增强。



8.5 构造函数



8.5.1 构造函数的作用

在例 8-1 讨论的类 `Date` 中,有一个专门负责初始化对象状态的函数 `InitDate()`。这个函数在对象创建后被显式地调用。这个步骤非常重要,否则,类对象将处在一个未知的状态,其后续的操作也将会得到未知的结果。但问题是,程序员有时会忘记调用初始化函数,或者调用了多次。这都是不好的。为此,有人希望初始化工作能这样进行:

```
class aClass
{
    int num = 0;    //在声明的时候初始化
    ...
};
```

但遗憾的是,这是错误的语法,因为 C++ 语言不允许在声明数据成员时进行初始化。

那么,又该如何将这项初始化工作自动化呢?

C++ 语言为类设计了构造函数 (constructor) 机制,它可以达到自动初始化数据成员的目的。

类的构造函数是类的一个特殊成员函数,它没有返回类型 (`void` 也不行),可以有参数,函数名和类名一样。

当创建类的一个新对象时,自动调用构造函数,完成初始化工作(需要注意构造函数是否有参数,以及参数的个数、类型)。

构造函数的作用为:

- ① 分配一个对象的数据成员的存储空间(该功能由系统自动完成);
- ② 执行构造函数(体),一般是初始化一个对象的部分或全体数据成员。

构造函数的函数体完成初始化对象的数据成员的任务,若希望该类所有对象的初始值相同,则构造函数可以不使用参数;若希望该类所有对象的初始值不相同,则通过使用带参数的构造函数可以做到这一点。

8.5.2 构造函数的定义

1. 构造函数的定义

考虑 8.1 节定义的 Circle 类，为它添加构造函数，如例 8-5 所示。

【例 8-5】 没有参数的构造函数。

```
class Circle
{
private:
    int x;
    int y;
    float fRadius;
public:
    Circle()    //构造函数
    {
        x = 0;
        y = 0;
    }
    void SetXY(int x, int y);
    void SetRadius(float r);
    void Move(int newx, int newy);
};
```

不带参数的构造函数将该类所有对象的圆心都初始化为原点。构造函数也可以带有参数，参见例 8-6。

【例 8-6】 带有默认参数的构造函数。

```
class Circle
{
private:
    int x;
    int y;
    float fRadius;
public:
    Circle(int xc = 0, int yc = 0)    //构造函数，参数可以省略
    {
        x = xc;  y = yc;
    }
    void SetXY(int x, int y);
    void SetRadius(float r);
};
```



```
void Move(int newx, int newy);  
};
```

该类的对象的圆心可以初始化为不同的位置。

2. 构造函数的初始化方式

构造函数有以下两种方式初始化数据成员。

- ① 在构造函数体内用赋值语句的方式；
- ② 用构造函数的初始化列表的方式。

构造函数的初始化列表是一种特殊的机制。它的语法格式如下：

构造函数名(参数列表) [: 成员名(表达式) [, 成员名(表达式)...]]

可以看出，这种初始化方式很像函数调用，它的作用是用表达式来初始化成员。下面是一个实例：

```
Circle(int xc, int yc) : x(xc), y(yc)    //等价于函数体中的 x=xc, y=yc  
{ }
```

需要注意的是，初始化列表的执行先于构造函数体的执行。并且，一些特殊成员，如常量成员和引用成员，都必须在初始化列表中进行初始化，参见例 8-7。

【例 8-7】 常量成员和引用成员的初始化。

```
struct structClass  
{  
    const int a;    //常量必须初始化，但不能在此处进行  
    const int & r;  //独立引用必须初始化，但不能在此处进行  
    //初始值表的方式，执行后常量 a 等于 9，r 是 a 的引用  
    structClass () : a(9), r(a) //初始化只能在这里进行  
    { }  
    /* structClass ()    //错误的构造函数  
    {  
        a = 9;          //错误的初始化，因为常量不能被赋值  
        r = a;          //这不是在进行引用绑定，而是在赋值  
    }  
    */  
};
```

3. 默认构造函数

用户定义的类型中，可以没有构造函数。此时编译器会自动给该类型生成一个没有参数的构造函数，该函数不做任何工作。这种构造函数称为默认构造函数。当建立一个对象时，这个默认构造函数被自动调用。不过，一旦一个类有一个显式定义的构造函数，哪怕这个函数是个空函数，编译器也不会代劳生成默认构造函数了。

但是，一个类型如果有 `const` 成员或引用成员，就不能使用默认构造函数。必须使用用户定义的构造函数进行初始化，而且只能使用表达式表的方式进行初始化。

4. 定义类对象时给构造函数提供参数

创建对象时，需要自动调用构造函数。如果构造函数有参数，那么，创建对象时必须给出对应的构造函数的实参，有两种方式。

◎ 只有一个参数：

类名 对象名 = 参数;

◎ 有一个或多个参数：

类名 对象名(参数列表);

构造函数带有参数的例子参见例 8-8。

【例 8-8】

```
class Date
{
private:
    int day, month, year;

public:
    Date(int d, int m, int y)
    {
        day = d; month = m; year = y;
    }
};

void main()
{
    Date today(1, 7, 2002);           //正确，调用构造函数
    Date startday(1, 1, 2001);        //正确，调用构造函数
    Date my_birthday;                 //错误，参数数目不够
    Date release1(10, 12);            //错误，参数数目不够
    Date release12(10, 12, "2002");   //错误，参数类型不对
}
```

8.5.3 重载构造函数

一个类可以提供多个构造函数，用于在不同场合进行类对象的初始化工作。很明显，这是构造函数的重载，它们的参数表必须互不相同。重载构造函数的例子参见例 8-9 和例 8-10。

【例 8-9】

```
class Date
{
private:
    int day, month, year;

public:
```



```
Date(int d, int m, int y);  
Date(int d, int m);  
Date(int d);  
Date();  
Date(const char * dateStr);  
};  
... // 构造函数实现
```

下述定义对象的方式都是正确的：

```
Date day1(14, 5, 2007);  
Date day2(14, 5);  
Date today(14);  
Date Christmas("Dec 25, 2007");  
Date now;
```

【例 8-10】

```
class Class_Name  
{  
public:  
    Class_Name();  
    Class_Name(int);  
    Class_Name(int, char);  
    Class_Name(float, char);  
};  
... // 构造函数实现  
void fun()  
{  
    Class_Name a;           // 调用构造函数 Class_Name();  
    Class_Name b(1);        // 调用构造函数 Class_Name(int);  
    Class_Name c(1, 'c');    // 调用构造函数 Class_Name(int,char);  
    Class_Name d(2.3, 'd');  //调用构造函数 Class_Name(float,char);  
}
```



8.6 复制构造函数

1. 复制构造函数定义

构造函数的参数可以是任何类型参数，甚至可以将自己类对象的（常量）引用作为参数，称之为复制构造函数。复制构造函数有两个含义：首先，它是一个构造函数，当创建一个新对象时，系统自动调用它；其次，它将一个已经定义过的对象（参数代表的对象）的数据成员逐一对应地复制给新对象。例 8-11 就是一个拥有复制构造函数的例子。

【例 8-11】

```

class X
{
public:
    X();
    X(const X& xObj); //复制构造函数，带有一个该类类型的对象引用参数
};

X a, b(a);           //b 调用自己的复制构造函数来复制 a

```

如果一个类没有显式定义复制构造函数，C++编译器可以为该类产生一个默认的复制构造函数。这个默认的复制构造函数采用 C 的方式，将复制对象的内存一个字节一个字节地复制到被复制对象的内存当中。这样一来，两个对象的内存映像就是一模一样的。

2. 复制构造函数的作用

复制构造函数的作用是：

- ① 创建一个新对象，并将一个已存在的对象复制给这个新对象；
- ② 对象的值作为参数（将实参对象复制给形参对象）；
- ③ 函数返回一个对象（复制返回的对象给一个临时对象）。

例 8-12 说明了复制构造函数的应用。

【例 8-12】复制构造函数的应用。

```

//Point.h
class Point
{
private:
    int x;
    int y;
public:
    Point(int intx = 0, int inty = 0)    //一般的构造函数
    {
        x = intx;
        y = inty;
    }
    Point(const Point &pt)    // 复制构造函数
    {
        x = pt.x;
        y = pt.y;
    }
    void Show()
    {
        cout << "x = " << x << ", y = " << y << endl;
    }
}

```



```
}
```

```
};
```

在程序中可以定义类对象：

```
Point a(10,20);      //调用 Point(int,int)
Point b(a);          //调用 Point(const Point &);
Point c = b;         //调用 Point(const Point &);
```

其中，`Point a(10,20);`创建了一个 `Point` 类的对象 `a`，初始参数为 10 和 20，它调用的是一般构造函数 `Point(int,int)`。而 `Point b(a);`创建一个 `Point` 类对象 `b`，并调用复制构造函数 `Point(const Point &)`，将对象 `a` 的数据成员逐域复制到新对象 `b` 中。`Point c = b;`创建了一个 `Point` 类对象 `c`，也调用复制构造函数，将对象 `b` 的数据成员逐域复制到新对象 `c` 中。

这是复制构造函数的典型使用，它表明当创建一个新对象，并希望将一个已存在的对象复制到这个新对象时，系统自动调用复制构造函数。

类对象的复制还发生在下面两种情况下：

- ◎ 对象本身作为参数；
- ◎ 函数返回对象。

对象的复制的例子参见例 8-13。

【例 8-13】

```
// Point.cpp
#include <iostream>
#include "Point.h"
Point fun(Point p)
{
    p.Show();
    return p;
}
void main()
{
    Point pt1;
    Point pt2 = fun(pt1);
    pt2.Show();
}
```

在 `main()`函数中，函数调用 `fun(pt1)`发生后，在堆栈上将生成一个临时对象，即形参对象 `p`，同时调用 `p` 的复制构造函数将实际参数 `pt1` 复制到其中。当从函数 `fun()`返回时，需要从栈上将 `p` 对象传送出来，这时 `pt2` 要调用一次自己的复制构造函数来复制 `p`。上述过程都是系统自动完成的。

3. 为什么需要复制构造函数

通常会忽略为一个类显式地提供复制构造函数。此时 `C++`编译器会为这个类产生一个默认的复制构造函数。默认的复制构造函数采用内存复制的形式，将已存在对象的内存一个

字节一个字节地复制到新建对象的内存当中。这样一来，新对象和老对象的内存映像就是一模一样的，可以说，新对象是老对象的“克隆体”。

在一般情况下，这个默认的复制构造函数工作得很好，但是，在一些特殊的情况下，它会出现问题。

例如，需要统计出一个类在程序运行过程中一共实例化出多少个对象。要完成这个工作，可以在类中加入一个静态变量，并且在该类的构造函数中自增该变量，然后在析构函数中减计数。很明显，如果计数以 0 开始，那么程序结束时它将会归 0。

【例 8-14】带计数器并且没有复制构造函数的类。

```
#include <iostream>
using namespace std;
class Point
{
private:
    int x;
    int y;
    static int nPtCount;           //用于对象计数
public:
    Point(int intx = 0, int inty = 0)    //一般的构造函数
    {
        x = intx;
        y = inty;
        nPtCount++;                //创建了对象，计数自增
        PrintCount("In constructor..."); //打印此时的计数值
    }
    ~Point()
    {
        nPtCount--;                //对象销毁，计数自减
        PrintCount("After object destroyed..."); //打印计数
    }
    static void PrintCount(const char *msg)
    {
        cout << msg << endl;
        cout << "Point count = " << nPtCount << endl;
    }
};
int Point::nPtCount = 0;
Point fun(Point p)    //注意这里，形参 p 的默认复制构造函数将被调用
{
```



```
        Point::PrintCount("In fun()...");
        return p;
    }
    void main()
    {
        Point pt1;
        Point pt2 = fun(pt1); //pt2 的默认复制构造函数将被调用
        pt2.PrintCount ("After point2 created...");
    }
```

这个程序的输出如下：

```
In constructor...
Point count = 1
In fun()...
Point count = 1
After object destroyed...
Point count = 0
After point2 created...
Point count = 0
After object destroyed...
Point count ==1
After object destroyed...
Point count ==2
```

很明显，输出的结果是错误的。代码似乎没有什么问题，那么错误究竟是在什么时候发生的呢？

注意输出的第 4 行：**Point count = 1**。这行输出已经有问题了，此时，内存中不仅存在着 **pt1** 对象，还有一个临时对象，即形参 **p**，它是由实参 **pt1** 复制得到的。复制的过程调用了 **p** 的默认构造函数（因为没有显式的复制构造函数）。我们已经知道，这个复制过程是内存的直接复制，没有任何代码被激活，所以计数器并没有增加而保持了原来的值。同样的情形也发生在函数 **fun()** 返回并将返回对象复制到 **pt2** 时。

解决问题的办法是，为 **Point** 类提供一个显式的复制构造函数。

【例 8-15】 为 **Point** 类提供显式的复制构造函数。

```
class Point
{
    //...
public:
    Point(const Point& p)                //显示的复制构造函数
    {
        x = p.intx;
```

```

        y = p.inty;
        nPtCount++;           //创建了对象，计数自增
        PrintCount("In copy constructor..."); //打印此时的计数值
    }
    //其他成员不变
};

```

这样一来，测试程序将会产生正确的输出（注释不是程序产生的）：

```

In constructor...
Point count = 1
In copy constructor...           //形参 p 复制实参 pt1 的结果
Point count = 2
In fun()...
Point count = 2
In copy constructor...           //pt2 接收 fun()返回值的结果
Point count = 3
After object destroyed...         //形参 p 销毁
Point count = 2
After point2 created...
Point count = 2
After object destroyed...         //pt2 销毁
Point count = 1
After object destroyed...         //pt1 销毁
Point count = 0

```

在 C++语言中，类是一种用户定义的类型，可能比较简单，也可能很复杂，系统难以把握。当类的对象本身作为函数参数、函数的返回值或者其他需要复制类对象时，系统很难提供一个统一的复制函数以适应各类对象的复制。因此，C++语言设置复制构造函数，不管复制构造函数是默认的，还是由用户提供的，它都属于一个具体的类。这样，当需要复制类对象时，C++调用这个类的复制构造函数。



8.7 析构函数

与构造函数对应的是析构函数。C++语言通过析构函数来处理对象的善后工作。

析构函数没有返回类型，没有参数，函数名是类名前加“~”。

析构函数的作用为：

- ◎ 执行析构函数（一般没有具体的工作）；
- ◎ 释放对象的存储空间（由系统自动完成）；
- ◎ 释放对象占用的资源（要由程序员设定）。

可以使用完全限定名方式显式地调用析构函数。若没有显式调用，则在一个对象的作用



域结束时，系统自动调用析构函数。析构函数的显示调用参见例 8-16。

【例 8-16】

```
class X
{
public:
    X() { }
    ~X() { }
};

void fun()
{
    X objX;
    ...
    objX.~X();    // 合法调用析构函数
    ...
}
```

上述这种显式调用的方式不常用。

在用户定义的类型中，可以没有析构函数，系统会自动给该类类型生成一个析构函数。该析构函数不做任何事情。但如果构造函数中使用 `new` 动态申请了存储空间，那么析构函数需要使用 `delete` 释放该空间，参见例 8-17。

【例 8-17】堆栈类。

```
//Stack.h
#include <iostream>
using namespace std;
const int MAXDEPTH = 1024;
class Stack
{
private:
    int SP;
    int nDepth;
    int * Cells;
public:
    enum OPCODE { OP_OVERFLOW, OP_ISEMPY, OP_SUCCESSFUL };
    Stack(int d = MAXDEPTH)
    {
        cout << "In constructor..." << endl;
        if ( d < 0 || d > MAXDEPTH ) d = MAXDEPTH;
        nDepth = d;
        SP = 0;
    }
}
```



```

        Cells = new int[d];
    }
    ~Stack()
    {
        cout << "In destructor..." << end;
        delete []Cells;
    }
}; ...

```

系统自动调用构造函数和自动调用析构函数的顺序是相反的。下面程序的输出可以证明这一点：

```

#include "Stack.h"
void main()
{
    Stack s(512);
}

```

这个程序的输出是：

```

In constructor...
In destructor...

```



8.8 对象的创建、释放和初始化

1. 对象的创建和释放

在 C++ 程序中，可以创建不同形式的各类对象。

(1) 命名的自动对象

每次进入该对象的作用域，都需要调用构造函数；每次退出该对象的作用域，都需要调用析构函数。析构函数的调用顺序参见例 8-18。

【例 8-18】

```

class Table {...};
void f(int a)
{
    Table aa;           //aa 的生命期到 f()函数运行结束
    Table bb;           //bb 的生命期到 f()函数运行结束
    if (a>0)
    {
        Table cc;       //cc 的生命期到 if 语句运行结束
        ...
    }
    Table dd;           //dd 的生命期到 f()函数运行结束
}

```



...

}

若调用函数 $f()$ ，则调用构造函数的顺序是： $aa \rightarrow bb \rightarrow cc \rightarrow dd$ ，调用析构函数的顺序是： $cc \rightarrow dd \rightarrow bb \rightarrow aa$ 。

(2) 自由对象（动态对象）

使用 `new` 创建对象（实际上是调用构造函数），使用 `delete` 释放对象（实际上是调用析构函数）。当使用 `delete` 释放对象后，该对象就不能再被使用。另外，如果构造函数有参数，也必须给出实参，参见例 8-19。

【例 8-19】动态对象的创建和释放。

```
//ex4_13.cpp
#include "Stack.h"
void main()
{
    Stack * pStack = new Stack(256);           //调用构造函数
    ...
    delete pStack;                             //调用析构函数
}
```

这个程序的输出是：

In constructor...

In destructor...

很遗憾的是，C++语言没有自动垃圾回收机制。所以，程序员必须记住自己曾经动态创建过对象，同时在合适的地方要销毁那些对象。否则，在程序结束后将会产生一个不引人注意的错误：内存泄露（Memory Leaks）。

2. 对象的初始化

初始化有许多表示法，C++语言允许下述 3 种表示方法。

(1) C 语言风格的初始值表的方法

```
struct conf
{
    char * month;
    int   year;
    char *   location;
}
cpp[] = { "Nov",      2001,   "SantaFe",
          "Oct",      2002,   "Denver",
          "Nov",      2006,   "Tyngsboro",
          "April",    2008,   "San Francisco"
};
```

这种方法适合对结构和数组进行初始化。

(2) 赋值表达式的方法

```
int num = 1;
char * p = "No.4.op.29";
```

这种方法适用于简单变量或指针类型变量的初始化。

(3) 表达式表的方法

在 C++ 语言中, 类的对象的数据成员的初始化一般通过构造函数进行。若构造函数有参数, 则在创建对象时, 必须给出实参 (即对象的初始值), 方法为:

```
Class_Name Object(...);
Class_Name Object=...;
```



8.9 对象和指针



8.9.1 this 指针

C++ 语言为所有非静态成员函数提供了一个称为 `this` 的指针, 因此, 常称成员函数拥有 `this` 指针。`this` 是一个隐含的指针, 不能被显式声明, 它只是一个形参, 一个局部变量, 它在任何一个非静态成员函数里都存在, 它局部于某一对象。

`this` 指针是一个常指针, 可以表示为 (但不能显式声明):

```
Class_Name * const this;
```

这里, `Class_Name` 是类名。因此, `this` 指针不能被修改和赋值。

某个对象 `obj` 调用某个成员函数 `fun()`, 则 `fun()` 函数的 `this` 指针就指向对象 `obj`, 而且在该成员函数 `fun()` 中, `this` 指针始终指向对象 `obj`。成员函数通过 `this` 指针, “知道”访问的数据是哪个对象的。参见例 8-20。

【例 8-20】

```
class INTEGER
{
private:
    int anInt;
public:
    void SetInt(int intNum)
    {
        anInt = intNum;
    }
};
```

实际上, 系统将上述程序改造为:

```
class INTEGER
{
```



```
private:
    int anInt;

public:
    void SetInt(int intNum, INTEGER * const this)
    {
        this->anInt = intNum;
    }
};
```

this 指针指示了当前成员函数工作在哪个具体的对象上。

this 指针是一个常指针，还可以使用 **const** 说明符将 **this** 声明为指向常量的常指针。在类中，一个成员函数的原型后跟一个 **const**，该函数称为 **const** 成员函数。它的特点是，该函数不能修改 **this** 所指的对象的成员。参见例 8-21。

【例 8-21】const 成员函数。

```
class INTEGER
{
private:
    int anInt;
public:
    void SetInt(int intNum)
    {
        anInt = intNum;
    }
    int GetInt() const
    {
        return anInt;
    }
};
```

GetInt() 函数后面的 **const** 说明此时 **this** 的类型是：

```
const INTEGER * const this
```

this 指针主要用于运算符重载、自引用等场合。例 8-22 给出 **this** 指针的一种典型应用。

【例 8-22】 在一个双向链表中添加一个结点。

```
class dlink          //将一个链表项说明为一个类
{
private:
    int    num;
    dlink  *pre;      //前驱
```

```

        dlink *suc;    //后继

public :
    dlink()
    {
        pre = NULL;
        suc = NULL;
    }
    void SetNum(int anint)
    {
        num = anint;
    }
    void Append (dlink * p);
};

void dlink::Append(dlink * p)
{
    p->suc = suc;      // p->suc = this->suc
    p->pre = this;     // 成员函数使用指向激活该类对象的 this 指针，称为自引用
    suc->pre = p;      // this->suc->pre = p
    suc = p;          // this->suc = p
}

dlink * list_head;    //头指针

void main()
{
    dlink d, d1, d2, *p1, *p2;
    d.SetNum(0);
    d1.SetNum(1);
    d2.SetNum(2);
    list_head = &d;
    p1 = &d1;
    p2 = &d2;
    list_head->Append(p1);
    list_head->Append(p2);
}

```

成员函数 `void Append (dlink * p)` 的功能是将 `p` 指向的结点添加到链表中，具体的位置是在激活该函数的结点的后面，而不是在整个链表的末尾，如图 8-3 所示。

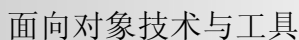


Diagram illustrating the initial state of a linked list:

A pointer labeled `List_head` points to a node. The node is a rectangle divided into three sections: the first section contains `NULL`, the second section contains `0`, and the third section contains `NULL`. Below the node, the letter `d` is centered.

```

graph LR
    List_head --> d
    subgraph d [d]
        direction LR
        d_data1[NULL] --- d_data2[0] --- d_next[ ]
    end
    subgraph d1 [d1]
        direction LR
        d1_prev[ ] --- d1_data1[1] --- d1_data2[NULL]
    end
    d_next --> d1_prev
    d1_prev --> d_next

```

```

graph LR
    List_head --> d
    subgraph d [d]
        direction LR
        d_data[0]
        d_next[ ]
    end
    subgraph d2 [d2]
        direction LR
        d2_prev[ ]
        d2_data[2]
        d2_next[ ]
    end
    subgraph d1 [d1]
        direction LR
        d1_prev[ ]
        d1_data[1]
        d1_next[ ]
    end
    d_data --- d_next
    d2_prev --- d2_data
    d2_data --- d2_next
    d1_prev --- d1_data
    d1_data --- d1_next
    d_next --> d2_prev
    d2_next --> d1_prev
    d1_next --> NULL
  
```

图 8-3 Append()函数将结点添加到链表中的位置

8.9.2 指向类对象的指针

在 C 语言中，结构分量可以通过结构变量直接存取或通过指向结构的指针来进行存取。类似地，在 C++ 语言中，可以直接使用对象，也可通过指向对象的指针变量来使用对象。

说明指向对象的指针变量的语法与说明指向其他任何数据类型的指针变量的语法相同：

```
One_Class_Name * obj_pointer;
```

说明 `obj_pointer` 是一个指向 `One_Class_Name` 类的对象的指针变量，因为没有创建对象，所以不需要调用构造函数。

可以通过以下方式:

指向对象的指针变量->类的公有成员

在类外访问一个类的成员。参见例 8-23。

【例 8-23】

```
// p_example.cpp
#include <iostream>
using namespace std;
class p_example
{
private:
    int num;
public:
    void SetNum(int val)
    {
        num = val;
    }
    void ShowNum()
```

```

    {
        cout << num << endl;
    }
};

void main()
{
    p_example ob;           //调用构造函数，创建对象
    p_example *p;           //声明指向对象的指针变量，没有调用构造函数
    ob.SetNum(1);
    ob.ShowNum();
    p = &ob;
    p->show_num();
}

```

输出为：

```

1
1

```

注意：对象 ob 的地址是用地址操作符获得的，这与获得任何其他类型变量地址的方法是相同的。

8.9.3 指向类的成员的指针

在 C++ 语言中，可以说明指向类的数据成员的指针和指向类的成员函数的指针。这两种指针必须与对象或指向对象的指针结合使用。

1. 指向类的数据成员的指针

指向类的数据成员的指针定义格式为：

类型名 类名::*指针；

说明指针只能指向指定类的指定类型的成员。

【例 8-24】指向类成员的指针。

```

#include <iostream>
using namespace std;
class INTEGER
{
private:
    int num1;
public:
    int num2;
    INTEGER(int anint)

```



```
{
    num1=anint;
    num2=0;
}
int GetInt() const
{
    return num1;
}
void OutInt( )
{
    cout << num1 << endl;
}
void Pointer()
{
    int INTEGER::*pnum1 = &INTEGER::num1;
    this->*pnum1 = 100;           //必须使用 this
}
};
void main()
{
    int INTEGER::*pnum2 = &INTEGER::num2;
    //    int INTEGER::*pnum=&INTEGER::num1; //错误，私有的
    INTEGER obj(10), * pobj;
    obj.OutInt();
    obj.Pointer();
    obj.OutInt();
    pobj = &obj;
    obj.*pnum2 = 1000;           //修改 obj 对象的 num2 成员
    cout << pobj->*pnum2;
}
```

输出为：

```
10
100
1000
```

2. 指向类的成员函数的指针

指向类的成员函数的指针定义的格式为：

类型名 (类名::*指针)(参数表);

函数指针并不属于类，而是只能指向类的指定原型的函数。

【例 8-25】指向成员函数的指针。

```

#include <iostream>
using namespace std;
class INTEGER
{
private:
    int num1;
public:
    int num2;
    INTEGER(int anint)
    {
        num1=anint;
        num2=0;
    }
    int GetInt() const
    {
        return num1;
    }
    void OutInt( )
    {
        cout << num1 << endl;
    }
    int Add(int anint)
    {
        return(num1 + anint);
    }
};

void main()
{
    INTEGER obj(12);
    int INTEGER::*pnum;
    pnum = &INTEGER::num2; //指向数据成员的指针
    obj.*pnum = 24;
    int (INTEGER::*pfun)(int); //指向成员函数的指针，该函数返回类型为 int，参数类型为 int
    pfun = INTEGER::Add;    //Add 成员函数符合要求
    INTEGER * pobj = &obj;
    cout << (pobj ->*pfun)(100) << endl;
}

```



```
cout << (obj.*pfun)(100) << endl;  
cout << obj.*pnum << endl;
```

```
}
```

输出为:

```
112  
112  
24
```



8.10 友元关系

一个对象的私有数据，只能通过成员函数进行访问，这是一堵不透明的墙。这种限制性的用法给两个类必须共享同一函数的情形带来了较大的开销。出于效率（而非技术上必需）考虑，C++语言提供了一种辅助手段，允许外面的类或函数去访问一个类的私有数据。

类 X 的友元 F 可以是一个函数，也可以是一个类，它虽然不是 X 的成员函数，但能访问 X 的私有成员和保护段的成员。除了具有这一访问权限外，F 在作用域、声明和定义等方面都是一普通的函数（或类）。



8.10.1 友元函数

在前面的 INTEGER 类中，num1 是个私有成员，所以在 INTEGER 类外，这个成员无法被直接访问。例如，函数：

```
void Print(const INTEGER& obj)  
{  
    cout << obj.num1 << endl; //错误，类外访问私有成员  
}
```

将会引起一个编译错误。为了使类外的函数能够访问一个类的私有数据，可以将该函数在类中声明为类的友元函数。参见例 8-26。

【例 8-26】

```
// INTEGER-4.cpp  
#include <iostream>  
using namespace std;  
class INTEGER  
{  
private:  
    int num1;  
public:  
    int num2;  
    INTEGER(int anint)
```

```

    {
        num1 = anint;
        num2 = 0;
    }
    friend void Print(const INTEGER& obj);
};

void Print(const INTEGER& obj)
{
    cout << obj.num1 << endl;
}

void main()
{
    INTEGER obj(12);
    Print(obj);
}

```

输出为:

12

通过友元函数，可以访问类的所有成员。友元函数的声明可以放在类的任何段里。友元函数不属于任何类，因此友元函数没有 `this` 指针，这是友元函数与成员函数的主要区别。

友元函数没有 `this` 指针，这与 `static` 成员函数类似，因此，在友元函数中要访问非 `static` 成员时，必须明确指出该友元函数在哪个对象上操作，并且必须有限定词（即需要通过对象或指向对象的指针进行），一般也就需要对象作为参数（除非是使用全局对象）。

有些时候，同样的功能可以由友元函数来完成，也可以由成员函数来完成，只是二者的参数和实现代码稍有区别，使用方式也有差别：友元函数必须在参数表中显式地指明要访问的对象，而成员函数“知道”在哪个对象上操作。所以，有些程序员喜欢友元函数所支持的通常的函数调用表示，不喜欢成员函数所需的成员调用表示，这种喜欢带有一种合理的偏见。但由于友元函数破坏了封装机制，所以建议尽量使用成员函数，只有在不得已的情况下，才使用友元函数。而且，现代的处理器的已经足够强大，效率问题可以通过处理器速度来弥补。针对于这个原则，例 8-25 可以改写为例 8-27。

【例 8-27】

```

#include <iostream>
using namespace std;
class INTEGER
{
private:
    int num1;
public:

```



```
int& Num1()           //定义一个访问函数。不考虑效率问题
{
    return num1;
}
//其他成员不变
};
void Print(const INTEGER& obj)
{
    cout << obj.Num1() << endl;
}
```

在一些面向对象的语言，比如 C#语言中，提供了一类属性函数的机制，它的实现类似于上面提到的访问函数。属性函数一般包括两个：**Get**（用于读取）和**Set**（用于设置）。这样，可以用更直接的方式访问类的私有成员。

8.10.2 友元类

可以将类 Y 声明为类 X 的友元（类），那么，类 Y 的所有成员函数都成为类 X 的友元函数。参见例 8-28。

【例 8-28】

```
#include <iostream>
using namespace std;
class Printer;
class INTEGER
{
private:
    int num1;
public:
    int num2;
    INTEGER(int anint)
    {
        num1 = anint;
        num2 = 0;
    }
    friend Printer;
};
class Printer
{
public:
```

```

void Print(const INTEGER& obj)
{
    cout << obj.num1 << endl;    //访问类的私有成员
}

void SetNum(INTEGER& obj, int anint)
{
    obj.num1 = anint;            //访问类的私有成员
}

};

void main()
{
    INTEGER obj(12);
    Printer ptr;
    ptr.SetNum(obj, 100);
    ptr.Print(obj);
}

```

输出为:

100

在上例中, `class Printer;` 是一个超前声明, 它引起编译器的注意, 表明名字 `Printer` 是一个类型名 (本例中是个类名)。当然, 可以去掉这个声明, 不过友元声明就必须改成:

```
friend class Printer;
```

同样的限制原则也适用于友元类。不要过多地使用友元, 因为这样会破坏数据封装原则。

8.10.3 友元关系的总结

1. 友元函数

友元函数不是类的成员函数, 是类外的函数, 所以, 它的声明放在类的私有段内或者公有段内是没有分别的。友元函数的调用也不需要通过对对象或指向对象的指针进行, 直接调用友元函数即可。

友元函数由于不是类的成员, 所以没有 `this` 指针, 这与 `static` 成员函数类似, 因此, 若在友元函数中要访问非 `static` 成员时, 必须明确指出该友元函数在哪个对象上操作, 必须要有限定词 (即需要通过对象或指向对象的指针进行), 一般也就需要对象做参数 (除非是使用全局对象)。

通过友元函数, 可以访问类的所有成员。

2. 友元类

若将类 `Y` 声明为类 `X` 的友元类, 则类 `Y` 的所有成员函数都成为类 `X` 的友元函数, 它们都能访问类 `X` 的所有成员。



友元机制的重要性在于两个方面：首先，某个函数可以是多个类的友元，使用友元函数能提高效率，使得表达简洁、清晰；其次，在运算符重载的某些场合需要使用友元。

友元类提供了不同类的对象之间合作的一种方式。类 A 是类 B 的友元类，那么，类 A 的对象访问类 A 的成员函数，这些函数又是类 B 的友元函数，它们可以访问类 A 和类 B 的所有成员，简单地说，类 A 的对象可以具有类 A 和类 B 的功能。

友元具有如下的特性：

- ◎ 非传递性，即 A 是 B 的友元，B 是 C 的友元，但 A 不一定是 C 的友元（除非将 A 声明为 C 的友元）；
- ◎ 非对称性，即 A 是 B 的友元，但 B 不一定是 A 的友元（除非将 B 声明为 A 的友元）。



8.11 与类和对象相关的问题



8.11.1 类类型作为参数类型

由于类是一个数据类型，也可以将对象作为参数传递给函数，参数传递遵循传值（或传地址）的方式，这与所有其他的数据类型是相同的。

类类型作为形参类型，一般有 3 种方式：

- ◎ 对象本身作为参数（传值）；
- ◎ 对象引用作为参数（传地址）；
- ◎ 对象指针作为参数（传值）。

具体例子参见例 8-29。

【例 8-29】

```
#include <iostream>
using namespace std;
class INTEGER
{
private:
    int num1;
public:
    INTEGER(int anint)
    {
        num2 = num1 = anint;
    }
    INTEGER(const INTEGER& obj)
    {
        cout << "In copy constructor..." << endl;
        num1 = obj.num1;
        num2 = obj.num2;
    }
}
```

```

    }
    int num2;
};

void Print(const INTEGER obj)    //传值
{
    cout << "In Print(INTEGER)..." << endl;
    cout << obj.num2 << endl;
    obj.num2 = 100;
}

void Printr(const INTEGER& obj)    //传引用, 注意这个函数名不是重载
{
    cout << "In Printr(INTEGER&)..." << endl;
    cout << obj.num2 << endl;
    obj.num2 = 200;
}

void Print(const INTEGER* pObj)    //传地址
{
    cout << "In Print(INTEGER*)..." << endl;
    cout << pObj->num2 << endl;
    obj.num2 = 300;
}

void main()
{
    INTEGER obj(1000);
    Print(obj);
    cout << obj.num2 << "\n";
    Printr(obj);
    cout << obj.num2 << "\n";
    Print(&obj);
    cout << obj.num2 << "\n";
}

```

输出为:

```

In copy constructor...
In Print(INTEGER)...
1000
100
In Printr(INTEGER&)...

```



```
1000
200
In Print(INTEGER*)...
200
300
```

下面是对程序的说明。

① **Print** 系列函数参数的 **const** 约束表明参数不能被修改。

② 如果一个函数有了传值参的版本，则不能同时有一个传引用的版本，因为编译器不能识别形如 **Print (obj)** 的调用是在调用哪个版本。所以，本例中的传引用版本的函数名为 **Printr**，而没有重载 **Print**。

③ 在传值调用版本中，实参对象传值给形参对象，是通过自动调用复制构造函数被完成的。这也是产生输出结果第 1 行的原因。

总之：

- ◎ 对象本身作为参数，对形参的任何修改都不影响实参的对象；
- ◎ 对象引用作为参数，对形参的任何修改就是对实参的对象的修改；
- ◎ 对象指针作为参数，对它指向的对象做任何修改就是对实参对象的修改。

8.11.2 一个类的对象作为另一个类的成员

一个类的对象可以作为另一个类的数据成员，简称为对象作成员，实现的是整体和部分之间的关系（a part of），即对象的包含关系，有时也称为类之间的“复合”（composition）。由于内部类对象“嵌埋”在外部类对象里，因此要访问内部类对象，必须“穿越”外部对象。一般的“穿越”做法是：在外部类中定义一些公有的访问函数，这些函数主要用于访问内部对象。这样的函数一般称为“包装函数”（wrapping function）。当然，也可以通过以下方式：

外部对象名.内部对象名.内部对象公有成员

来直接访问内部对象的公有成员。参见例 8-30。

【例 8-30】

```
class Date
{
public:
    int day, month, year;
    //...
};
class Person
{
public:
    Date DOB;
    char Name[50];
```



```

    char Gender;
    //...

    int& Year()    //包装函数
    {
        return DOB.year;
    }
    //...
};

class Class
{
public:
    Person students[50];
    //...
};

class Grade
{
public:
    Person    administrator;
    Class classes[5];
    //...
};

```

上面的例子示意了复合的几种方式：

- ◎ 一个类对象作为另一个类的成员，如 **Person**（人）类包含 **Date** 类的对象；
- ◎ 一个类的多个对象作为另一个类的成员，如 **Class**（班级）类包含 **Person** 类的数组；
- ◎ 多个类的对象作为另一个类的成员，如 **Grade**（年级）类包含 **Person** 类的对象和 **Class** 类的数组。

一个复合类在构造的时候，首先调用的是内部类对象的构造函数。如果内部类对象多于一个，则它们的构造函数的调用顺序依照它们的定义顺序而定，此后，再调用外部类的构造函数。而在复合类对象析构的时候，析构函数的调用顺序正好与构造顺序相反。

8.11.3 临时对象

当函数的返回类型为类类型时，将调用复制构造函数把返回的对象保存到那个临时对象中。另外，在类对象的运算中，也可能产生临时对象。临时对象被视为常量对象，意思是，该对象的所有数据成员都不能改变。

临时对象也可以由显示构造函数的调用来创建，参见例 8-31。

【例 8-31】

```

class A
{

```



```
public:
    A() { /*...*/ }
    A(const A& o) { /*...*/ }
    //...
};
A fun(A obj)           //形参 obj 就是一个临时对象
{
    return obj;         //返回的对象被复制到一个临时对象中
}
void main()
{
    A a = fun(A()); //函数的参数是一个通过调用构造函数创建的临时对象
}
```

习题 8

8.1 编写一个杂凑表（Hash Table）类，以字符串作为关键字存放和查找记录。提供公共成员函数进行杂凑表的记录插入、查询和删除。

8.2 对于一个可移植的编译器，如何把编译器中的公共部分从依赖机器的部分中分离出来？能否将与该编译过程中依赖机器的部分界面表示为一个抽象数据类型（类）？

8.3 tnode 的结构如下：

```
struct tnode
{
    char * tword;
    int count;
    tnode * left,    *right;
};
```

把 tnode 设计为带有构造函数和析构函数的类。要求构造函数使用 new 为 tnode 指向的串分配空间，析构函数用 delete 回收已分配的空间。

8.4 在 C 语言或 C++ 语言中，块（BLOCK）定义为用 “{” 和 “}” 括起来的程序段，局部变量或局部对象的作用域就是定义它们的块。使用构造函数和析构函数实现记录进入块和退出块的运行时间轨道。

8.5 为提供均匀分布的随机数定义一个类，该类含有构造函数用来指定分布参数，用函数 draw 返回“下一个”值。

第 9 章 C++语言实现多态性

多态性是指一个运算符或一个名字具有多种含义。

在面向对象的 C++语言中，多态是通过重载（或称为超载 **overload**）来实现的。

面向对象的程序设计语言中允许两种方式的重载：函数名重载和运算符重载。

本章介绍运算符重载。



9.1 重载运算符

一个类就是用户定义的一个类型。因此，类类型可以作为参数类型，可以作为返回类型；可以定义一个类的单个对象，也可以说明对象数组，甚至还可能有类类型常量。

运算符表达简单的操作，很直观，但只能对于语言定义的类型（的数据）进行操作。

`data1+data2*data3` 的表示就比 `multiply data2 by data3 and add the result to data1` 直观得多。

对于用户定义的类型，某些操作也想使用运算符来表示，则需要对运算符进行重载。

在基本数据类型上，系统提供了许多预定义的运算符，它们以一种简捷的方式工作。例如“+”运算符：

```
int num1, num2;
num2 = num1 + num2;
```

表达的是两个整数相加，很简捷。但是，两个串类的对象的合并：

```
class String
{
public:
    String string_cat(String);
    ...
};
String str1, str2, str3;
str3 = str1.string_cat(str2);
```

表达起来就不如 `str3 = str1 + str2` 那样简捷。

为了表达上的方便，希望已预定义的运算符，也可以在特定类的对象上以新的含义进行解释。例如，在 `string` 类对象 `str1`, `str2` 的环境下，运算符“+”能被解释为串 `str1` 和 `str2` 的合并。换言之，希望预定义运算符能被重载，使得某个类的对象能够直接使用运算符来表示对对象的操作。

C 语言中，有许多系统预定义的运算符，如“+”，它可以用于 `int` 类型数据，也可用于 `float` 类型数据。虽然使用相同的运算符，但生成的目标代码不相同，因为整数和浮点数在



内存中的表示是不同的。这时，“+”运算符具有两种不同的解释（实现代码）。也就是说，像“+”这样的运算符在C语言中已经被重载。不幸的是，C语言仅支持少量有限的运算符重载。C++语言扩充了这个功能，允许已存在的预定义运算符由用户在不同的上下文中做出不同的解释。即：类类型的对象使用运算符，使用的是运算符新的含义；而其他类型的数据使用运算符，使用的是运算符原来的含义。

在原来预定义的运算符的含义的基础上，再定义对于某个用户定义类型的对象进行操作的新的含义。这就是运算符重载。

运算符重载后，优先级和结合性不变。

对复数的“加法”操作参见例9-1。

【例9-1】

```
#include <iostream.h>

class Complex
{
    double re,im;
public:
    Complex(double r,double i):re(r),im(i){ }
    Complex(){ re=0;im=0;}
    Complex add_Complex(Complex);
    ...
};

Complex Complex::add_Complex(Complex cobj)
{
    Complex temp;
    temp.re=re+cobj.re;
    temp.im=im+cobj.im;
    return temp;
    //整个函数体也可简化为 return Complex(re+cobj.re, im+cobj.im);
}

void main( )
{
    Complex obj1(1,2),obj2(3,4);
    Complex obj3=obj1.add_Complex(obj2);
    ...
}
```

对于两个复数相加，定义成员函数 `add_Complex (Complex)` 实现该操作，使用

`obj1.add_Complex(obj2);`

表示该函数的调用，不太直观。能不能将两个复数相加表示为：`obj1+obj2` 呢？

C++语言提供了一种将标准定义的运算符用在用户自定义类型上的方法，称为运算符重

载。C++语言约定，如果一个成员函数的函数名字是特殊的，即由关键字 `operator` 加上一个运算符构成，如 `operator+`，那么，`obj1+obj2` 就表示该函数的隐式调用，即等价于 `obj1.operator+(obj2)` 的显式调用。

因此，`obj1+obj2` 称为函数 `operator+(...)` 的隐式调用形式；

`obj1.operator+(obj2)` 称为函数 `operator+(...)` 的显式调用形式；

函数 `operator+(...)` 称为运算符重载函数。

重载“+”的复数类参见例 9-2。

【例 9-2】

```
#include <iostream.h>

class Complex
{
    double re,im;
public:
    Complex(double r,double i):re(r),im(i){ }
    Complex( ){ re=0;im=0;}
    Complex operator+(Complex );
    ...
};

Complex Complex ::operator+(Complex cobj)
{
    Complex temp;
    temp.re=re+cobj.re;
    temp.im=im+cobj.im;
    return temp;
}

void main( )
{
    Complex obj1(1,2),obj2(3,4);
    Complex obj3=obj1+obj2;// Complex obj3=obj1.operator+(obj2)
    ...
}
```

上述程序中的 `obj1+obj2` 等价于 `obj1.operator+(obj2)`。

通过以上例子，可以看到重载运算符的好处。本来 C++语言提供的运算符只能用于 C++语言的标准类型数据结构的运算，但 C++语言程序设计的重要基础是类和对象，允许用户自己定义新的类型。如果 C++语言的运算符都无法用于类对象，对于类对象的赋值运算、数值运算、关系运算、逻辑运算和输入/输出操作，只能通过普通函数调用方式进行，不能直接用运算符形式进行处理，则类和对象的应用将会受到很大限制。如果为了解决这个问题，C++语言为用户自定义的类对象另外提供一批功能相似的新的运算符，则用户需要记忆更多



的运算符及其使用规则，因此 C++ 语言允许重载现有的运算符，使这些简单易用、众所周知的运算符能够直接作用于用户自定义的类对象，扩大了运算符的作用范围。

9.1.1 运算符重载的语法形式

在 C++ 语言中，运算符通过一个运算符重载函数进行重载。运算符重载可以采用成员函数和普通函数两种重载方式，语法格式如下。

(1) 运算符重载函数为成员函数

语法格式为：

```
type Class_Name::operator@(参数表)
{
    //相对于 Class_Name 类而定义的操作
}
```

其中，type 是返回类型，“@”是要重载的运算符符号，Class_Name 是重载该运算符的类的类名，函数名 operator@ 是关键字 operator 后跟要重载的运算符符号“@”。这里用“@”泛指能被重载的运算符。

(2) 运算符重载函数为友元函数

语法格式为：

```
type operator@(参数表)
{
    //相对于该类而定义的操作
}
```

运算符分为一元和二元运算符，一元运算符需要一个操作数，二元运算符需要两个操作数。参数表中罗列的就是该运算符所需要的操作数。

运算符函数体对重载的运算符的含义做出新的解释。这里所解释的含义只与重载该运算符的类（即类 Class_Name）有关。在 Class_Name 类对象的上下文中，该运算符的含义由这个函数体进行解释（或称执行）；否则，该运算符具有系统预定义的含义。因此，当一个运算符被重载时，它所有原先的含义并未失去，只是定义了相对于某特定类（这里是 Class_Name）的一个新运算符。参见例 9-3。

【例 9-3】

```
#include <iostream.h>
class Complex
{
    double re,im;
public:
    Complex(double r,double i):re(r),im(i){ }
    Complex(){ re=0;im=0;}
    Complex operator+(Complex );
```

```

        void print();
    };
    Complex Complex ::operator+(Complex  cobj)
    {
        Complex  temp;
        temp.re=re+cobj.re;
        temp.im=im+cobj.im;
        return temp;
    }
    void Complex::print()
    {
        if(im>0)
            cout<<re<<"+"<<im<<"i"<<endl;
        else
            cout<<re<<im<<"i"<<endl;
    }
    void main( )
    {
        Complex obj1(1,2),obj2(3,4);
        Complex obj3=obj1+obj2;    //即 Complex obj3=obj1.operator+(obj2);
        obj3.print();
        int i1=5,i2=9,i3;
        i3=i1+i2;
        cout<<i3<<endl;
    }

```

程序中, `obj3=obj1+obj2` 的 “+” 调用的是 `Complex` 重载的 `operator+` 运算符, 而 `i1+i2` 调用的是系统预定义的运算符, 各自按照各自的运算规则进行处理。

9.1.2 重载运算符规则

(1) 大多数系统预定义的运算符可以通过运算符重载函数定义它们对用户定义类型进行操作的新的含义, 只有少数的 C++ 语言的运算符不能重载, 例如:

- :: (域运算符)
- sizeof (长度运算符)
- ?: (条件运算符)
- * (仅作为前缀使用, 不能重载)
- . (成员访问运算符)
- .* (成员指针访问运算符)



域运算符和 `sizeof` 运算符的运算对象是类型而不是变量或者一般表达式，不具备重载的特征，而成员访问运算符和成员指针访问运算符不能重载是为了保证访问成员的功能不能被改变。

(2) 只能对已有的 C++ 运算符进行重载，C++ 语言允许用户自己定义新的运算符，如 “\$” 和 “**” 等。对于不是运算符的符号，如 “;” 等是不能重载的，C++ 语言不允许的运算符也不能重载。

(3) 重载运算符时，不能改变它们的优先级，不能改变它们的结合性，也不能改变这些运算符所需操作数的数目。

(4) 重载运算符的函数不能有默认的参数，否则就改变了运算符所需要的操作数的数目。

(5) 重载的运算符必须和用户自定义的类对象一起使用，其参数至少应有一个是类对象（或类对象的引用）。也就是说，参数不能全部是 C++ 语言的标准类型，以防止用户修改用于标准类型数据的运算符的性质。参见例 9-4。

【例 9-4】

```
#include <iostream>
using namespace std;
int operator +(int a,int b)
{
    return (a-b);
}
int main()
{
    int x=10,y=5;
    cout<<x+y<<endl;
}
```

上述程序的运行结果到底是按照 C++ 语言的 “+” 运算符的标准定义显示 15 呢？还是按照重载运算符显示 5？这样会使得用户修改预定义运算符的含义，因此是禁止的。

(6) 用于类对象的运算符一般必须重载，只有两个运算符：赋值运算符 “=” 和地址运算符 “&” 可以不必重载。

赋值运算符 “=” 可以在同类对象之间相互赋值。系统为每一个新声明的类默认重载了一个赋值运算符，逐个赋值类的的数据成员。当系统提供的默认赋值函数不能满足用户需求时，就需要用户按照自己对该类的赋值需求，自己重载赋值运算符。

地址运算符 “&” 能够返回类对象在内存中的起始地址，因此也不必重载。

(7) 重载运算符时，可以让运算符执行任意的操作，例如，将 “+” 运算符重载为 “-” 运算，将 “>” 运算符重载为 “<” 运算符。但这样违背了运算符重载的初衷，降低程序可读性，使人无法理解程序功能。因此，应当使重载运算符的功能类似于该运算符作用于标准类型数据时所实现的功能。如果不能建立运算符的这种习惯用法，应该采用函数调用方法，以免造成阅读困难。

9.1.3 一元运算符和二元运算符

重载运算符的函数一般定义为类的公有的非 `static` 的成员函数或类的友元函数。

虽然也可以使用 `static` 成员函数重载运算符，但不能隐式调用，失去了运算符重载函数的意义。如果使用类外的非成员函数（也非友元函数）重载运算符，由于不能直接访问类的私有成员，一般只能对必须包含有公有数据的类进行，或者，要受到某些条件的限制，所以一般不采用。

运算符重载时，运算符重载函数一般定义为两种方式，或者作为一个类的成员函数，或者作为友元函数。这两种方式非常类似，但还有许多差别，其关键原因是，成员函数具有 `this` 指针，而友元函数没有 `this` 指针。

1. 一元运算符

一元运算符，不论前缀还是后缀，都需要一个操作数。现在，暂时不区分前缀和后缀的重载，9.1.4 节通过“++”运算符详细解释前缀和后缀重载的区别。

对一元运算符@，两种重载方式说明如下。

(1) 成员函数重载运算符

```
type Class_Name::operator@()
{
    ...
}
```

设 `obj` 为 `Class_Name` 的类对象。

显式调用方式：

```
obj.operator@()
```

隐式调用方式：

```
@obj 或者 obj@
```

一元运算符重载函数 `operator @` 所需的一个操作数通过 `this` 指针隐含地传递，因此，它的参数表为空。

(2) 友元函数重载运算符

```
type operator@(Class_Name obj)
{
    ...
}
```

显式调用方式：

```
operator@(obj)
```

隐式调用方式：

```
@obj 或者 obj@
```

一元运算符重载函数 `operator@` 所需的一个操作数在参数表中。

无论运算符@重载为类的成员函数还是友元函数，隐式调用方式都是相同的。参见



例 9-5 和例 9-6。

【例 9-5】

```
#include <iostream>
using namespace std;
class Complex
{
    double re,im;
public:
    Complex(double r,double i):re(r),im(i){ }
    Complex( ){ re=0;im=0;}
    Complex operator!();
    void print();
};
Complex Complex::operator!()
{
    Complex temp;
    temp.re=-re;
    temp.im=-im;
    return temp;
}
void Complex::print()
{
    cout<<re;
    if(im>0)cout<<"+"<<im<<"i"<<endl;
    else cout<<im<<"i"<<endl;
}
void main( )
{
    Complex obj(1,2);
    obj.print();
    obj=!obj;
    obj.print();
}
```

【例 9-6】

```
#include <iostream.h>
class Complex
{
    double re,im;
```

```

public:
    Complex(double r,double i):re(r),im(i){ };
    Complex( ){ re=0;im=0;}
    friend Complex operator!(const Complex &obj)
    void print();
};
Complex operator!(const Complex &obj)
{
    Complex temp;
    temp.re=-obj.re;
    temp.im=-obj.im;
    return temp;
}
void Complex::print()
{
    cout<<re;
    if(im>0)cout<<"+"<<im<<"i"<<endl;
    else cout<<im<<"i"<<endl;
}
void main( )
{
    Complex obj(1,2);
    obj.print();
    obj=!obj;
    obj.print();
}

```

例 9-5 和例 9-6 分别用类的成员函数和类的友元函数重载了一元运算符“!”，函数隐式调用方式都为!obj。类的成员函数原型为 `Complex operator!()`，函数没有参数，使用的参数通过 `this` 指针传递。类的友元函数原型为 `friend Complex operator!(const Complex &aobj)`。

`!obj` 表达式是对 `obj` 对象求反。为了减少参数传递过程中的内存分配次数和调用复制构造函数，提高程序运行效率，参数以引用的方式传递；同时表达式并不会修改 `obj` 对象的值，为了保护传递到函数内部的 `obj` 对象，将其声明为 `const`，可以保护其不会在函数内部被修改，因此参数设置为常引用 `const Complex &obj` 是最佳的声明方式。当然，如果简单声明为 `Complex obj`，程序也能够正确运行，但形参和实参结合的时候要增加一次复制构造函数的调用，多分配一次 `Complex` 类型的存储空间，降低效率，因此不推荐采用。

`!obj` 表达式的值可以保存起来，而且应该也是 `Complex` 类型的，因此声明函数类型为 `Complex` 类型，由于函数值返回的是函数内部的局部对象，函数返回时局部对象会被销毁，因此只能复制局部对象的值到函数对象中（函数对象是一个临时对象），而不能声明函数的



返回值为引用 (`Complex & operator!(...)`)。表达式的值可能会被修改 (如 `(!obj)++`) 因此不能被 `const` 修饰。

2. 二元运算符

二元运算符，需要两个操作数。

对二元运算符@，两种重载方式说明如下。

(1) 成员函数重载运算符

```
type Class_Name::operator@(Class_Name obj2)
{
    ...
}
```

显式调用方式：

```
obj1.operator@(obj2)
```

隐式调用方式：

```
obj1 @ obj2
```

二元运算符重载函数 `operator @` 所需的第一个操作数通过 `this` 指针隐含地传递。第二个操作数通过参数提供，因此，它只有一个参数。

(2) 友元函数重载运算符

```
type operator @(Class_Name obj1,Class_Name obj2)
{
    ...
}
```

显式调用方式：

```
operator @(obj1,obj2)
```

隐式调用方式：

```
obj1 @ obj2
```

二元运算符重载函数 `operator @` 所需的两个操作数都通过参数提供，因此，它有两个参数。不管是用成员函数还是友元函数重载运算符，该运算符的隐式调用方法是相同的。参见例 9-7 和例 9-8。

【例 9-7】

```
include <iostream>
using namespace std;
class Complex
{
    double re,im;
public:
    Complex(double r,double i):re(r),im(i){ }
    Complex( ){ re=0;im=0;}
    Complex operator+(const Complex &obj);
```

```

        Complex operator!();
        void print();
};
Complex Complex::operator+(const Complex &obj)
{
    Complex temp;
    temp.re=re+obj.re;
    temp.im=im+obj.im;
    return temp;
}
Complex Complex::operator!()
{
    Complex temp;
    temp.re=-re;
    temp.im=-im;
    return temp;
}
void Complex::print()
{
    cout<<re;
    if(im>0)cout<<"+"<<im<<"i"<<endl;
    else cout<<im<<"i"<<endl;
}
void main()
{
    Complex obj1(1,2),obj2(3,4);
    Complex obj3=obj1+!obj2;// Complex obj3=obj1.operator+(! obj2)
    obj3.print();
}

```

【例 9-8】

```

#include <iostream.h>
class Complex
{
    double re,im;
public:
    Complex(double r,double i):re(r),im(i){ }
    Complex( ){ re=0;im=0;}
    friend Complex operator+(const Complex &aobj,const Complex &bobj);

```



```
friend Complex operator!(const Complex &aobj);
void print();
};
Complex operator+(const Complex &aobj,const Complex &bobj)
{
    Complex temp;
    temp.re=aobj.re+bobj.re;
    temp.im=aobj.im+bobj.im;
    return temp;
}
Complex operator!(const Complex &obj)
{
    Complex temp;
    temp.re=-obj.re;
    temp.im=-obj.im;
    return temp;
}
void Complex::print()
{
    cout<<re;
    if(im>0)cout<<"+"<<im<<"i"<<endl;
    else cout<<im<<"i"<<endl;
}
void main()
{
    Complex obj1(1,2),obj2(3,4);
    Complex obj3=obj1+!obj2;// Complex obj3=obj1.operator+(obj2)
    obj3.print();
}
```

用成员函数重载运算符和用友元函数重载运算符，它们传递参数的方法不一样，也就导致了它们的实现代码不相同。

注意：重载双目运算符为友元函数时，如果使用了命名空间，Visual C++ 6.0 编译器提示出错信息：fatal error C1001: INTERNAL COMPILER ERROR。这是因为有的编译器系统（如 Visual C++ 6.0）没有完全实现 C++语言标准，命名空间不支持这种重载方式。因此，要么把该重载函数改成成员函数，要么不使用命名空间，可将

```
#include <iostream>
using namespace std
```

这两行修改为：

```
#include <iostream.h>
```

3. 运算符重载为成员函数和友元函数的选择建议

有时，用成员函数重载运算符会碰到麻烦，参见例 9-9。

【例 9-9】

```
#include <iostream>
using namespace std;
class Complex
{
    double re,im;
public:
    Complex(double r=0,double i=0):re(r),im(i){ }
    Complex operator+(const Complex &obj);
    Complex operator!();
    void print();
};
Complex Complex ::operator+(const Complex &obj)
{
    Complex temp;
    temp.re=re+obj.re;
    temp.im=im+obj.im;
    return temp;
}
Complex Complex::operator!()
{
    Complex temp;
    temp.re=-re;
    temp.im=-im;
    return temp;
}
void Complex::print()
{
    cout<<re;
    if(im>0)cout<<"+"<<im<<"i"<<endl;
    else cout<<im<<"i"<<endl;
}
void main()
{
```



```
Complex obj1(1,2),obj2(3,4);
Complex obj3=obj1+obj2;           // Complex obj3=obj1.operator+(obj2)
obj3.print();

obj3=obj1+27;
obj3.print();
obj3=27+obj1;                     //错误, 27 不是 Complex 的对象
obj3.print();
}
```

编译提示有错误, `obj3=obj1+27` 的显式调用是 `obj3=obj1.operator+(27)`, 但 `Complex` 类没有将整数 27 转换为 `Complex` 类的函数, 因此提示出错。修改例 9-9 为例 9-10。

【例 9-10】

```
#include <iostream.h>
class Complex
{
    double re,im;
public:
    Complex(double r=0,double i=0):re(r),im(i){ }
    friend Complex operator+(const Complex &obj1,const Complex &obj2);
    Complex operator!();
    void print();
};
Complex operator+(const Complex &obj1,const Complex &obj2)
{
    Complex temp;
    temp.re=obj1.re+obj2.re;
    temp.im=obj1.im+obj2.im;
    return temp;
}
Complex Complex::operator!()
{
    Complex temp;
    temp.re=-re;
    temp.im=-im;
    return temp;
}
void Complex::print()
{
```



```

        cout<<re;
        if(im>0)cout<<"+"<<im<<"i"<<endl;
        else cout<<im<<"i"<<endl;
    }
    void main( )
    {
        Complex obj1(1,2),obj2(3,4);
        Complex obj3=obj1+!obj2;           // Complex obj3=obj1.operator+(obj2)
        obj3.print();
        obj3=obj1+27;
        obj3.print();
        obj3=27+obj1;
        obj3.print();
    }

```

由于 `obj3=obj1+27` 的参数 27 通过调用有一个参数默认的构造函数 `Complex(double r,double i=0)`，系统可以自动将 27 转换为 `Complex(27)`对象，因此编译通过。

但 `obj3=27+obj1` 的显式调用为 `obj3=27.operator+(obj1)`，而 27 不是 `Complex` 类型，因此编译器不能调用 `Complex` 的 `operator+()`重载运算符完成加法操作。将例 9-10 的 `operator+()`运算符重载为友元函数，如果两个输入参数中任何一个为整数或者实数，可以调用类的有一个参数默认的构造函数 `Complex(double r,double i=0)`自动转换为 `Complex` 对象，因此 `obj3=27+obj1` 或者 `obj3=obj1+27` 的调用可用。如果两个参数都是实数，则调用系统预定义的实数加法完成加法运算。

由于成员函数仅能为一个“实际对象”所调用，而友元函数无此限制，因此，若运算符的操作需要修改类对象的状态，则它应该是成员函数，而不应是友元函数。需要左值操作数的运算符（如 `=`，`+=`，`++`）的重载最好用成员函数；相反，如果运算符所需的操作数（尤其是第一个操作数）希望有隐式类型转换，则该运算符重载必须用友元函数，而不是成员函数。

许多人偏爱友元函数的表达风格，宁肯用友元函数的表达方式，也不愿用成员函数的表达方法。但对于某些特殊的情况，选择成员函数比较好。成员函数调用语法使用户清楚地知道对象是否可修改，而友元函数采用的参数在这方面是非常不明显的；成员函数能隐含 `this` 指针，而友元函数必须至少带一个显式参数，成员函数方式表达较为简捷。用户可根据具体情况进行选择。另外，友元函数破坏了封装机制，建议尽量使用成员函数实现操作。

注意：“`=`”、“`()`”、“`[]`”、“`->`”不能使用友元函数进行重载，其余的运算符都可以使用友元函数来实现重载，而流插入运算符“`<<`”和流提取运算符“`>>`”只能使用友元函数来实现重载。

9.1.4 重载“++”和“--”的前缀和后缀方式

在 C 语言中，运算符“++”（或“--”）有两种方式，前缀方式 `++variable` 和后缀方式



variable++。

早期的 C++ 语言在重载运算符 “++”（或 “—”）时，不能显式地区分是前缀形式还是后缀形式。在目前的 C++ 语言标准中，对此做了一些约定。

(1) 前缀方式 ++obj

成员函数重载：

```
Class_Name Class_Name::operator++( );
```

友元函数重载：

```
Class_Name operator++(Class_Name &);
```

(2) 后缀方式 obj++

成员函数重载：

```
Class_Name Class_Name::operator++(int);
```

友元函数重载：

```
Class_Name operator++(Class_Name &,int);
```

这时，第 2 个参数（int）只是一个占位符号，用来区分该重载函数是前缀方式还是后缀方式，在函数内部不需要也不能使用。显式调用时，该占位符号通常用 0 表示。

例如，obj++等价于 obj++(0)或者 obj++=0。

【例 9-11】 使用成员函数，区分一元运算符的前缀和后缀。

```
#include <iostream>
using namespace std;
class Complex
{
    double re,im;
public:
    Complex(double r=0,double i=0):re(r),im(i){ }
    Complex operator+(const Complex &obj);
    Complex & operator++(); //前缀方式
    Complex operator++(int); //后缀方式
    Complex operator!();
    void print();
};
Complex Complex ::operator+(const Complex &obj)
{
    Complex temp;
    temp.re=re+obj.re;
    temp.im=im+obj.im;
    return temp;
}
Complex & Complex::operator++()
```

```
{
    re++;
    im++;
    return *this;
}

Complex Complex::operator++(int x)
{
    Complex c(*this);//复制构造函数
    re++;
    im++;
    return c;
}

Complex Complex::operator!()
{
    Complex temp;
    temp.re=-re;
    temp.im=-im;
    return temp;
}

void Complex::print()
{
    cout<<re;
    if(im>0)cout<<"+"<<im<<"i"<<endl;
    else cout<<im<<"i"<<endl;
}

void main( )
{
    Complex obj1(1,2),obj2;
    cout<<"first obj1=";
    obj1.print();
    obj2=obj1++;
    cout<<"second obj1=";
    obj1.print();
    cout<<"and obj2=";
    obj2.print();
    obj2=++obj1;
    cout<<"third obj1=";
    obj1.print();
}
```



```
        cout<<"and obj2=";  
        obj2.print();  
    }
```

类似地，也可以使用友元函数进行重载，参见例 9-12。

【例 9-12】使用友元函数重载，区分一元运算符的前缀和后缀。

```
#include <iostream>  
using namespace std;  
class Complex  
{  
    double re,im;  
public:  
    Complex(double r=0,double i=0):re(r),im(i){ }  
    Complex operator+(const Complex &obj);  
    friend Complex & operator++(Complex & obj);//前缀方式  
    friend Complex  operator++(Complex & obj,int);//后缀方式  
    Complex operator!();  
    void print();  
};  
Complex Complex ::operator+(const Complex  &obj)  
{  
    Complex  temp;  
    temp.re=re+obj.re;  
    temp.im=im+obj.im;  
    return temp;  
}  
Complex & operator++(Complex & obj)  
{  
    obj.re++;  
    obj.im++;  
    return obj;  
}  
Complex  operator++(Complex & obj,int x)  
{  
    Complex c(obj);//复制构造函数  
    obj.re++;  
    obj.im++;  
    return c;  
}
```

```

Complex Complex::operator!()
{
    Complex temp;
    temp.re=-re;
    temp.im=-im;
    return temp;
}

void Complex::print()
{
    cout<<re;
    if(im>0)cout<<"+"<<im<<"i"<<endl;
    else cout<<im<<"i"<<endl;
}

void main( )
{
    Complex obj1(1,2),obj2;
    cout<<"first obj1=";
    obj1.print();
    obj2=obj1++;
    cout<<"second obj1=";
    obj1.print();
    cout<<"and obj2=";
    obj2.print();
    obj2=++obj1;
    cout<<"third obj1=";
    obj1.print();
    cout<<"and obj2=";
    obj2.print();
}

```

程序的运行结果为：

```

first obj1=1+2i
second obj1=2+3i
and obj2=1+2i
third obj1=3+4i
and obj2=3+4i

```

由于“++”运算符修改类对象本身，因此在友元调用方式中，为了能够修改传递进函数的参数，以对象引用的方式传递。“++”的前缀方式返回类型为类对象的引用，后缀方式返回类型是类对象，是因为前缀方式返回的是变化之后的类对象，该类对象是作为对象引用



传递进函数的，在函数调用结束之后不会被销毁，因此可以返回类的对象的引用，以便节约内存空间和提高效率；而“++”的后缀方式，函数的返回值是参数对象被修改之前的值（放到局部对象中暂时存储），因此返回的是局部对象的值，而局部对象在函数返回的时候会被销毁，因此函数的返回值只能将局部对象的值复制过来，而不能直接使用局部对象空间，故后缀方式返回类型为类对象，而不能是类对象的引用。

重载运算符“—”也用类似的方法。

重载运算符“++”（或“—”）后，可以用隐式调用和显式调用两种方式来使用，对于后缀方式，也可以有非 0 的实参。

例如，显式调用：

```
obj.operator++(3);
```

或

```
operator++(obj,3);
```

是合法的，它们等价于 `obj++=3` 或者 `obj++(3)`。

9.1.5 重载赋值运算符

赋值运算符“=”可以被重载，用户可以定义自己需要的重载“=”的运算符重载函数。重载了的运算符函数 `operator=` 不能被继承，而且它必须被重载为成员函数。一般重载格式为：

```
X X::operator=(const X & from)
{
    //复制 X 的成员
}
```

当用户在一个类中显式地重载了运算符“=”时，称用户定义了类赋值运算。它将一个 X 类对象 `from` 逐域复制到赋值号左端的类对象中。

类的赋值运算符跟其他的运算符略有不同。如果用户没有为一个类重载赋值运算符，编译程序将生成一个默认的赋值运算符。赋值运算把源对象逐域复制到目的对象中。对许多简单的类，如 `Complex` 类，默认的赋值函数工作得很好。

复制构造函数和赋值运算符都是把一个对象的数据成员复制到另一对象中，而且两者的配备看来都有点奇怪。它们的区别是，复制构造函数要创建一个新对象，而赋值运算符则是改变一个已存在的对象的值。

对于绝大多数重要的类，有了默认赋值运算符也许仍不够，需要用户自己进行重载，当默认类赋值函数不能正确工作时，应当考虑这一问题。参见例 9-13。

【例 9-13】

```
#include <iostream>
#include <assert.h>
using namespace std;
class cstring
{
```

```
public:
    cstring(char * str="");
    cstring(const cstring & str);
    cstring operator+(const cstring & str);
    cstring & operator=(const cstring &str);
    void print();

private:
    char * m_pstr;
    int m_ysize;
};

cstring::cstring(char * str)
{
    if(str==NULL)
    {
        m_ysize=0;
        m_pstr=NULL;
    }
    else
    {
        m_ysize=strlen(str);
        m_pstr=new char[m_ysize+1];
        assert(m_pstr);
        strcpy(m_pstr,str);
    }
}

cstring::cstring(const cstring & str)
{
    if(str.m_ysize==0)
    {
        m_ysize=0;
        m_pstr=NULL;
    }
    else
    {
        m_ysize=str.m_ysize;
        m_pstr=new char[m_ysize];
        assert(m_pstr);
        strcpy(m_pstr,str.m_pstr);
    }
}
```



```
    }  
}  
cstring cstring::operator+(const cstring & str)  
{  
    cstring s1;  
    s1.m_ysize=m_ysize+str.m_ysize-1;  
    s1.m_pstr=new char[s1.m_ysize];  
    assert(s1.m_pstr);  
    strcpy(s1.m_pstr,m_pstr);  
    strcpy(&(s1.m_pstr[m_ysize]),str.m_pstr);  
    return s1;  
}  
cstring & cstring::operator=(const cstring &str)  
{  
    if(&str==this)  
        return *this;  
    if(m_ysize>0)  
        delete []m_pstr;  
    m_ysize=str.m_ysize;  
    m_pstr=new char[m_ysize];  
    assert(m_pstr);  
    strcpy(m_pstr,str.m_pstr);  
    return *this;  
}  
void cstring::print()  
{  
    cout<<m_pstr<<endl;  
}  
void main()  
{  
    cstring s1("hello"),s2(" every one!"),s3;  
    s3=s1+s2;  
    s3.print();  
}
```

所有的赋值符号（包括复合赋值符号）都会修改左值。因此，左值不能是被 `const` 修饰的：如果是友元函数，则参数的左值不能被 `const` 修饰；如果是成员函数，则函数不能是 `const` 函数。为了能够连续赋值，应该返回左值的引用，以便进行修改。尽管连续赋值是从左往右读的，但编译器是从右往左解析的，因此赋值符号的返回值是否是 `const` 无所谓，都可以支

持连续赋值。但如果希望赋值的同时调用成员函数，如(a=b).func()，先将 b 对象赋值给 a 对象，再调用 a 的成员函数，这个函数可能对 a 对象进行处理，但如果赋值语句的返回值被 const 修饰，则可能不能修改 a。因此，赋值操作符应该对左值返回非 const 的引用。

9.1.6 重载运算符 “()” 和 “[]”

运算符 “()” 和运算符 “[]” 不能用友元函数重载，只能采用成员函数重载。

1. 重载函数调用运算符()

对应的运算符重载函数为 operator()(…)，设 obj 为类 Class_Nam 的一个对象，则表达式 obj(arg1,arg2)可被解释为 obj.operator()(arg1,arg2)，参见例 9-14。

【例 9-14】

```
#include <iostream>
#include <assert.h>
using namespace std;
class cstring
{
public:
    cstring(char * str="");
    cstring(const cstring & str);
    cstring operator+(const cstring & str);
    cstring & operator=(const cstring &str);
    int operator()();
    void print();
private:
    char * m_pstr;
    int m_ysize;
};
cstring::cstring(char * str)
{
    if(str==NULL)
    {
        m_ysize=0;
        m_pstr=NULL;
    }
    else
    {
        m_ysize=strlen(str)+1;
        m_pstr=new char[m_ysize];
```



```
        assert(m_pstr);
        strcpy(m_pstr,str);
    }
}

cstring::cstring(const cstring & str)
{
    if(str.m_ysize==0)
    {
        m_ysize=0;
        m_pstr=NULL;
    }
    else
    {
        m_ysize=str.m_ysize;
        m_pstr=new char[m_ysize];
        assert(m_pstr);
        strcpy(m_pstr,str.m_pstr);
    }
}

cstring cstring::operator+(const cstring & str)
{
    cstring s1;
    s1.m_ysize=m_ysize+str.m_ysize-1;
    s1.m_pstr=new char[s1.m_ysize];
    assert(s1.m_pstr);
    strcpy(s1.m_pstr,m_pstr);
    strcpy(&(s1.m_pstr[m_ysize-1]),str.m_pstr);
    return s1;
}

cstring & cstring::operator=(const cstring &str)
{
    if(&str==this)
        return *this;
    if(m_ysize>0)
        delete []m_pstr;
    m_ysize=str.m_ysize;
    m_pstr=new char[m_ysize];
    assert(m_pstr);
```

```

        strcpy(m_pstr,str.m_pstr);
        return *this;
    }
    int  cstring::operator()()
    {
        return m_ysize;
    }
    void cstring::print()
    {
        cout<<m_pstr<<endl;
    }
    void main()
    {
        cstring s1("hello"),s2(" every one!"),s3;
        s3=s1+s2;
        s3.print();
        cout<<s3()<<endl;
    }

```

程序运行结果：

hello every one!

17

2. 重载下标运算符“[]”

相应的运算符重载函数为 `operator[](…)`，设 `xobj` 为类 `X` 的对象，则表达式 `xobj[arg]` 解释为 `xobj.operator[](arg)`，参见例 19-15。

【例 9-15】

```

#include <iostream>
#include <assert.h>
using namespace std;
class cstring
{
public:
    cstring(char * str="");
    cstring(const cstring & str);
    cstring operator+(const cstring & str);
    cstring & operator=(const cstring &str);
    int operator()();
    char operator[](int i);
    void print();

```



```
private:
    char * m_pstr;
    int m_isize;
};

cstring::cstring(char * str)
{
    if(str==NULL)
    {
        m_isize=0;
        m_pstr=NULL;
    }
    else
    {
        m_isize=strlen(str)+1;
        m_pstr=new char[m_isize];
        assert(m_pstr);
        strcpy(m_pstr,str);
    }
}

cstring::cstring(const cstring & str)
{
    if(str.m_isize==0)
    {
        m_isize=0;
        m_pstr=NULL;
    }
    else
    {
        m_isize=str.m_isize;
        m_pstr=new char[m_isize];
        assert(m_pstr);
        strcpy(m_pstr,str.m_pstr);
    }
}

cstring cstring::operator+(const cstring & str)
{
    cstring s1;
    s1.m_isize=m_isize+str.m_isize-1;
```

```

        s1.m_pstr=new char[s1.m_ysize];
        assert(s1.m_pstr);
        strcpy(s1.m_pstr,m_pstr);
        strcpy(&(s1.m_pstr[m_ysize-1]),str.m_pstr);
        return s1;
    }
cstring & cstring::operator=(const cstring &str)
{
    if(&str==this)
        return *this;
    if(m_ysize>0)
        delete []m_pstr;
    m_ysize=str.m_ysize;
    m_pstr=new char[m_ysize];
    assert(m_pstr);
    strcpy(m_pstr,str.m_pstr);
    return *this;
}
int cstring::operator()()
{
    return m_ysize;
}
char cstring::operator[](int i)
{
    if(i>=m_ysize)
        return '\0';
    else
        return m_pstr[i];
}
void cstring::print()
{
    cout<<m_pstr<<endl;
}
void main()
{
    cstring s1("hello"),s2(" every one!"),s3;
    s3=s1+s2;
    s3.print();
}

```



```
cout<<s3()<<endl;
cout<<"which character are display?input the place: "<<endl;
int place;
cin>>place;
cout<<place<<" character is "<<s3[place]<<endl;
}
```

如果输入 3，则程序运行结果为：

```
hello every one!
17
which character are display? input the place:
3
3 character is l
```

9.1.7 重载输入运算符和输出运算符

在标准文件 `iostream.h` 中，有两个标准的类类型：`istream` 和 `ostream`。`istream` 类将运算符“>>”重载为输入运算符，它对系统预定义类型（如 `int`，`char`，`float`，`double` 等类型）进行重载。`ostream` 类将运算符“<<”重载为输出运算符，它也对系统预定义类型重载。对于预定义类型，用户可以方便地使用运算符“>>”和“<<”进行输入和输出。对于类类型，用户可以重载运算符“>>”和“<<”以满足自己的需要。

输出运算符“<<”的第一个操作数是 `cout`，它实际上是标准类类型 `ostream` 的对象的引用（它的定义在文件 `iostream.h` 中），若在程序中，用户自己定义一个 `ostream` 的对象的引用 `s`，则 `s` 也可以使用运算符“<<”。

例如：

```
#include <iostream.h>
void main()
{
    int num=10;
    ostream & scout=cout;
    scout<<num<<endl;
}
```

则输出

```
10
```

对某个用户定义的类型 `Class_Name` 重载输出运算符“<<”，重载函数的函数名为 `operator<<`，而且只能使用友元函数进行重载，因为隐式调用方式为：

```
cout<< Class_obj;
```

其中 `Class_obj` 是类类型 `Class_Name` 的对象。

如果将显式调用方式解释为：

```
cout.operator<<(Class_obj);
```

那么, `cout` 应该是类类型 `Class_Name` 的对象, 但 `cout` 是类类型 `ostream` 对象的引用, 不是 `Class_Name` 的对象。

为了保证输出运算符 “<<” 的连用性, 重载函数的返回应该为 `ostream &`。

重载函数有两个参数: `ostream & stream` 和 `Class_Name obj`, 分别对应实参 `cout` 和 `Class_obj`。

利用友元函数重载了输出运算符 “<<”, 就可以直接输出一个对象。

类似地, 可以利用友元函数重载输入运算符 “>>”, 但要注意第 2 个参数必须是对象的引用。

参见例 9-16, 修改复数类 `Complex` 的 `print` 函数为 `operator<<`, 并增加输入函数。

【例 9-16】 复数类的声明放到头文件 `Complex.h` 中。

```
#include <iostream.h>
class Complex
{
    double re,im;
public:
    Complex(double r,double i=0):re(r),im(i){ }
    Complex( ){ re=0;im=0;}
    Complex operator+(const Complex &obj);
    Complex operator!();
    friend ostream & operator<<(ostream & os,const Complex & c);
    friend istream & operator>>(istream & is,Complex & c);
};
```

复数类的实现放到 `Complex.cpp` 中:

```
#include <iostream.h>
#include "Complex.h"
Complex Complex::operator+(const Complex &obj)
{
    Complex temp;
    temp.re=re+obj.re;
    temp.im=im+obj.im;
    return temp;
}
Complex Complex::operator!()
{
    Complex temp;
    temp.re=-re;
    temp.im=-im;
```



```

        return temp;
    }
    ostream & operator<<(ostream & os,const Complex & c)
    {
        os<<c.re;
        if(c.im>0) os<<"+"<<c.im<<"i"<<endl;
        else os<<c.im<<"i"<<endl;
        return os;
    }
    istream & operator>>(istream & is,Complex & c)
    {
        is>>c.re>>c.im;
        return is;
    }

```

测试程序调用复数类实现两个复数的求和并显示打印结果，并测试“>>”功能。测试程序 `Complex -main.cpp` 如下：

```

#include "Complex.h"
void main( )
{
    Complex obj1(1,2),obj2(3,4);
    Complex obj3=obj1+!obj2;    // Complex obj3=obj1.operator+(obj2)
    cout<<obj3;

    cin>>obj3;
    cout<<obj3;

}

```

9.1.8 指针悬挂问题

指针悬挂是指使用 `new` 申请的存储空间无法访问，也无法释放。造成指针悬挂原因是，对指向 `new` 申请的存储空间的指针变量进行赋值修改。

标准类型的指针悬挂程序参见例 9-17。

【例 9-17】

```

#include <string.h>
void main( )
{
    char * p,* q;

```



```

    p=new char[10];
    q=new char[10];
    strcpy(p, "abcd");
    q=p;
    delete [] p;
    delete [] q;
}

```

语句 `q=p;` 将 `q` 和 `p` 指向相同的地址, 而 `q` 原来指向的空间再也无法访问, 并且也不能够释放, 如图 9-1 所示, 这就是指针悬挂。语句 `delete [] q;` 还会导致两次释放同一块存储空间 (`delete p` 已经释放过一次)。

对象的复制有两种方式: 初始化和赋值。在将一个对象的数据成员对应地赋值给另一个对象的数据成员过程中, 如果一个类包含有指针类型的数据成员, 那么, 该类的两个对象的复制就可能会有问题, 因为对象包含的指针成员如果直接赋值, 就可能导致指针悬挂。解决的方法为, 重新定义复制构造函数和重载 “=” 的函数。

例如, 自定义一个字符串类, 用私有指针动态分配存储区存储字符串, 参见例 9-18。

【例 9-18】

```

class String
{
    char * pstr;
    int sz;
public:
    String (int s)
    {
        pstr=new char [sz=s];
    }
    ~String ( )
    {
        delete [] pstr;
    }
};

void main( )
{
    String str1(10);
    String str2(10);
    str2=str1;
}

```

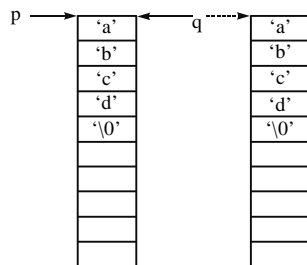


图 9-1 指针悬挂



执行该程序，会出现运行错误，如图 9-2 所示。

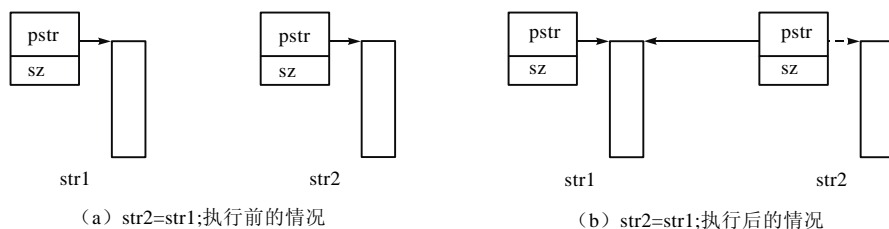


图 9-2 String 类的指针悬挂

默认类赋值函数 `str2=str1` 是将对象 `str1` 的数据成员（即字符指针 `pstr` 和整型变量 `sz`）逐个复制到 `str2` 中，这样 `str2` 中原先的值就丢失了，它原先指向的内容被封锁起来而没办法再用，这就产生了指针悬挂问题。更为严重的是，由于 `str1` 和 `str2` 中的 `pstr` 都指向同一内存，当 `str1` 和 `str2` 都退出其所在的作用域，即 `main()` 结束时，这块内存被释放了两次（调用了两次析构函数），这是一个非常严重的错误。

解决这一问题的方法是，定义 `String` 类对象的赋值要恰当，需要重载 “=” 运算符为新的语义，还需要考虑复制构造函数的情况。

原来默认的复制构造函数为：

```
String::String(const String & str)
{
    pstr = str.pstr;
    sz = str.sz ;
}
```

当

```
String s2=s1;
```

使得 `s2` 和 `s1` 的指针指向相同的空间（即 `s1` 的指针指向的空间）。

原来默认的重载 “=” 函数为：

```
String String ::operator=(const String & str)
{
    pstr = str.pstr;
    sz = str.sz ;
}
```

修改例 9-18 为例 9-19。

【例 9-19】

```
#include <string.h>
class String
{
    char * pstr;
    int sz;
```

```

public:
    String (int s)
    {
        pstr=new char [sz=s];
    }
    ~String ( )
    {
        delete [ ] pstr;
    }
    String(const  String & );
    String operator=(const String &);
};

String::String(const String & str)
{
    pstr=new char[sz=str.sz] ;
    strcpy(pstr,str.pstr);
}

String String ::operator=(const String & str)
{
    if (this!=&str)                                //防止自己赋值给自己
        return  *this;
    delete[ ] pstr;
    pstr=new char[sz=str.sz];
    strcpy(pstr,str.pstr);
    return *this;
}

或者

String String ::operator=(const String & str)
{
    char *pstr1=new char[sz=str.sz];
    strcpy(pstr1,str.pstr);
    delete[ ] pstr;
    pstr=pstr1;
    return *this;
}

void main( )
{
    String str1(10);

```



```
String str2(10);  
str2=str1;  
}
```

这样就解决了指针悬挂的问题。当在构造函数中使用 `new` 进行动态存储分配，在析构函数中调用 `delete` 进行动态回收时，程序员一定要注意考察默认的类的拷贝和重载“=”函数是否能正常工作。



9.2 C++语言的类型转换

类型转换是指将一种类型的值转换为另一种类型的值。对于类类型，是否也存在一种类型转换机制，使得类对象之间能进行类型转换？一般来说，类对象类型的转换可由构造函数和转换函数说明。这种转换常称为用户定义的类型转换或类类型转换。在 C++ 语言中，类被视为用户定义的类型，可以像系统预定义类型一样进行类型转换。

C++ 语言允许的类型转换有 4 种：

- ◎ 标准类型 → 标准类型
- ◎ 标准类型 → 类类型
- ◎ 类类型 → 标准类型
- ◎ 类类型 → 类类型

标准类型包括除 `class`、`struct` 和 `union` 类型（即所有的类类型）外的其他所有类型。对于标准类型，C++ 语言提供了以下两种类型转换：

- ◎ 隐式类型转换
- ◎ 显式类型转换

1. 隐式转换

隐式转换发生在下述的情况下：

- ① 混合运算，级别低的向级别高的转换；
- ② 将表达式的值赋给变量，表达式的值向变量类型的值转换；
- ③ 实参向形参传值，实参的值向形参的值进行转换；
- ④ 函数返回结果，返回的值向函数返回类型的值进行转换。

2. 显式类型转换

显式类型转换方式有以下两种。

- ① 强制法：

(类型名) 表达式

或者

(类型名)(表达式)

- ② 函数法：

类型名 (表达式)

它们都将表达式强制地转换为类型名所代表的类型的值。

9.2.1 标准类型转换为类类型

可以通过自定义的重载赋值号“=”的函数和构造函数实现标准类型→类类型的转换，它们都需要有标准类型的参数。

具有标准类型参数的构造函数说明了一种从参数类型到该类类型的转换，参见例 9-20。

【例 9-20】

```
#include <string.h>
class INTEGER
{
    int num;
public:
    INTEGER (int i);
    INTEGER (const char * str);
    void mem_fun(INTEGER anint);
};
INTEGER::INTEGER(int i)
{
    num=i;
}
INTEGER::INTEGER(const char *str)
{
    num=strlen(str);
}
void INTEGER::mem_fun(INTEGER anint)
{
    num=anint.num;
}
void main()
{
    INTEGER obj1= INTEGER(1);
    INTEGER obj2="ChengDu";
    int anint=10;
    INTEGER obj3= INTEGER(anint);
    obj1=20;           //obj1.operator=(INTEGER(20));
    obj2.mem_fun(3);    // obj2.mem_fun(INTEGER(3));
}
```

语句 `INTEGER obj1= INTEGER(1);` 将 1 转换为类类型 `INTEGER` 后赋给对象 `obj1`。执行



语句 `INTEGER obj2="ChengDu";` 时, 编译尝试用构造函数 `X(const char *)` 对赋值号右边的字符串进行类类型转换, 转换成功后, 赋给 `INTEGER` 的对象 `obj2`。语句 `obj2.mem_fun(3);` 中, 由于函数 `mem_fun` 需要一个 `INTEGER` 的对象作为参数, 因此尝试用构造函数对实参进行转换, 转换成功后, 进行虚实参数匹配, 执行函数调用。这样的转换是系统自动进行的, 称为隐式类型转换。构造函数 `INTEGER(int)` 将整数类型转换为类类型 `INTEGER`, 构造函数 `INTEGER(const char *)` 将字符串转换为类类型 `INTEGER`。

当 `INTEGER` 的构造函数不进行这个转换时, 该转换失败。亦即, 系统不再作其他转换的尝试。

例如:

```
class X
{
    ...

public:
    X(int);
    ...
};

class Y
{
    ...

public:
    Y(X);
    ...
};

void main()
{
    X objX=1;
    Y objY=objX;
    Y a=1;           //错误
    ...
}
```

类 `Y` 没有构造函数 `Y(int)`, 因此不进行转换, 但它不会再去尝试转换 `Y(X(1))`, 系统不这样去做。

程序员要想保证不会发生不期望的隐式转换, 可以采用例 9-21 的方式进行处理。

【例 9-21】 创建构造函数放在私有段的类的对象中。

```
class vector
{
public:
    static vector make (int s)
```

```

    {
        vector v = s;
        return v;
    }
    friend vector make1(int s)
    {
        vector v = s;
        return v;
    }
private:
    vector (int s); //构造函数放在私有段
};
void f()
{
    vector v1=vector::make(10);
    vector v2=make1(10);
    v1=10;           // 错误：私有构造函数，类外不能调用
    v1=vector::make(99);
}

```

构造函数 `vector(int s)` 放在私有段中，并在公有段设置一个 `static` 函数或设置类外的一个友元函数来调用这个构造函数。当出现语句 `v1=10;` 时，它需要用构造函数隐式地转换，但构造函数是私有的，无法进行，因此避免了不期望的隐式转换。

9.2.2 类类型转换函数

带一个参数的构造函数可以进行类型转换，但是它的转换功能很受限制。

例如，想将一个类转换为基本类型就办不到。因此，需要引入一种特殊的成员函数——类型转换函数，它在类对象之间提供一种类似显式类型转换的机制。

C++语言允许程序员为类定义一个类型转换函数，它的语法格式如下：

```

Class_Name::operator type()
{
    ...
    return (type 类型的实例);
}

```

类型转换函数没有参数，没有返回类型，但这个函数体内必须有一条返回语句，返回一个 `type` 类型的实例。

类型转换函数不能被重载，因为它没有参数。

类型转换函数的功能是将 `Class_Name` 类型的对象转换为类型为 `type` 的实例。`type` 可以



是一个预定义类型，也可以是一个用户定义的类型。

类型转换函数只能定义为一个类的成员函数，而不能定义为类的友元。

类类型转换为整型，参见例 9-22。

【例 9-22】

```
#include <iostream.h>
class INTEGER
{
    int num;
public:
    INTEGER(int anint=0)
    {
        num=anint;
    }
    operator int( )
    {
        return num;
    }
};
void main( )
{
    INTEGER obj(12);
    int anint=int(obj);
    cout<<anint<<endl;
    anint=(int)obj;
    cout<<anint<<endl;
    anint=obj;                //anint=obj.operator int( );
    cout<<anint<<endl;
}
```

输出为：

```
12
12
12
```

主函数中出现的 3 种情况，所赋的值都是由 `INTEGER::operator int()` 来进行转换的。语句 `int anint=int(obj);` 和 `anint=(int)obj;` 从形式上看都是用类型的强制类型转换规则，其实不然，因为 `obj` 是一个类对象，它使用 `int(obj)` 或 `(int)obj` 的表达形式，实际执行为：

```
obj.operator int( )
```

语句 `anint=obj;` 也用该类型转换函数进行。

除了赋值和初始化，类型转换函数还可以这样用：


```

void fun(INTEGER num1, INTEGER num2)
{
    int anint1=(num1)?(1+num1):0;
    int anint2=(num1&&num2)? num1+num2 : anint1;
    if (num1)
    {
        ...
    }
}

```

可见, `INTEGER` 的对象 `num1`、`num2` 可直接当做整型数据使用。系统总是尝试将 `num1` 和 `num2` 具有的类型按类类型转换函数转换为 `int` 类型。这是一种隐式类型转换方式。

类型转换函数有两种使用方式:

隐式使用 `anint=obj;`
 显式使用 `anint=obj.operator int();`

一般用隐式方式, 当需要明确指出用哪一个类型转换函数时, 才使用显式方式。

至此, 已知道, 语句 `int anint=obj;` 用类型转换函数进行转换, 而语句 `INTEGER obj=anint;` 用构造函数进行转换。但是如果一个类既有用于转换的构造函数, 又拥有类型转换函数, 如:

```

INTEGER(int);
operator int( );

```

那么, 语句 `obj=obj+anint;` 该如何解释? 这里, 可以将表达式 `obj+anint` 解释为:

```
obj.operator int( )+anint
```

即, 将 `obj` 转换得到一个整型数, 与 `anint` 相加后得到一个整型结果, 再用 `INTEGER(int)` 将结果转换为 `INTEGER` 的类型, 赋给 `obj`。

也可以这样解释: `obj+INTEGER(int)` 将 `anint` 转换为一个 `INTEGER` 类型, 与 `obj` 相加后再将结果赋给 `obj`。

这里存在二义性, 编译不知道用哪一种方式进行解释。用户在这种情况下必须显式地使用类型转换函数。

例如:

```

INTEGER obj1=anint;
obj=obj+obj1;           //两个对象加

```

或

```
obj=(int)obj+anint;      //两个整数加
```

用户定义的类型转换函数, 只有在它们无二义性时才能隐式地使用。

类类型转换为类类型, 参见例 9-23。

【例 9-23】

定义一个类 `integer`, 它可以处理 32 位的整数; 同时定义另一个类 `real`, 它可以处理 32 位的实数。

```
#include <iostream.h>
```



```
class real;
class integer
{
    long lval;
public:
    integer()
    { lval=0; }
    integer(long d)
    { lval=d; }
    operator real();          //类型转换函数
    long & operator()( )
    { return lval; }
    friend integer operator+(integer num1,integer num2);
    integer operator-(integer num1);
    friend ostream & operator<<(ostream &s,integer num);
};
class real
{
    float rval;
public:
    real() { rval=0; };
    real(float d) { rval=d; };
    operator integer(); //类型转换函数
    float & operator()( ) { return rval; };
    friend real operator+(real num1,real num2);
    real operator-(real num1);
    friend ostream & operator<<(ostream &s,real num);
};
```

类 `integer` 的一个无参构造函数将它的数据成员 `lval` 置为 0, 而构造函数 `integer(long)` 将一个长整型对象转换为类 `integer` 的一个对象。同样, 类 `real` 的一个构造函数将它的数据成员 `rval` 置为 0, 而构造函数 `real(float)` 将一个浮点对象转换为 `real` 的一个对象。

运算符 “+” 被重载为一个友元, 而运算符 “-” 被重载为一个成员函数, 以说明它们对类型转换机制的影响。运算符 “()” 被重载为成员函数, 它的功能是返回数据成员 `val` 的存储单元。类 `integer` 和类 `real` 都定义了一个类型转换函数, 类 `integer` 的转换函数将类 `integer` 的对象转换为 `real` 类型, 类 `real` 的转换函数将它的对象转换为 `integer` 类型。

下面是这两个类的成员函数的实现:

```
integer::operator real()
{
```

```

    real num;
    num( )=(float) lval;
    return num;
}

```

在这个类型转换函数中，说明了一个具有 `real` 类型的自动对象 `num`。`num()` 返回 `num` 对象的内部数据成员 `rval` 的存储单元。语句 `num()=(float) lval;` 将 `integer` 类对象的内部数据 `lval` 通过标准类型转换为一个浮点类型，并将转换结果赋给 `num` 的内部数据 `rval`。`return` 语句将这个 `num` 对象返回，从而完成了一个类类型的转换。

```

integer operator+(integer num1,integer num2)
{
    integer num;
    num.lval=num1.lval +num2.lval;
    return num;
}
integer integer::operator-(integer num1)
{
    integer num;
    num.lval=lval-num1.lval;
    return num;
}
ostream & operator<<(ostream &s,integer num)
{
    return (s<<num.lval);
}

```

下面的类型转换函数将 `real` 类型转换为 `integer` 类型。

```

real::operator integer( )
{
    integer n;
    n( )=(long) rval;
    return n;
}

```

下面的类型转换函数将具有 `real` 类型的对象转换为一个具有 `integer` 类型的对象。

```

real operator+(real num1,real num2)
{
    real num;
    num.rval=num1.rval+num2.rval;
    return num;
}

```



```
real real::operator-(real num1)
{
    real num;
    num.rval=rval-num1.rval;
    return num;
}

ostream & operator<<(ostream &s,real num)
{
    return (s<<num.rval);
}
```

下述测试例子给出了这两个类的应用。

```
void main( )
{
    integer i1;
    i1=50;
    cout<<"\n i1="<<i1;
    real r1=56.6;
    cout<<"\n r1="<<r1;
    integer i2;
    i2=r1;          // i2.operator=(r1.operator integer( ))
    cout<<"\n i2=r1,i2="<<i2;
    real r2;
    r2=i1;
    cout<<"\n r2=r1,r2="<<r2;
    real r3;
    r3=real(i1)+r1;
    cout<<"\n r3=real(i1)+r1,r3="<<r3;
    r3=i1-r1;
    cout<<"\n r3=i1-r1,r3="<<r3;
    integer i3;
    i3=i1+integer(r3);
    cout<<"\n i3=i1+integer(r3),i3="<<i3;
    i3=i1-r3;
    cout<<"\n i3=i1-r3,i3="<<i3;
    integer i4=24.5;
    cout<<"\n i4=24.5,i4="<<i4;
    real r4=40;
    cout<<"\n r4=40,r4="<<r4;
```

```

        cout<<"\n";
    }

```

该程序的输出为：

```

i1=50
r1=56.6
i2=r1,i2=56
r2=r1,r2=50
r3=real(i1)+r1,r3=106.6
r3=i1-r1,r3=-6
i3=i1+integer(r3),i3=44
i3=i1-r3,i3=56
i4=24.5,i4=24
r4=40,r4=40

```

上述这个测试程序，主要用于说明类型转换运算如何进行。

语句 `r3=i1+r1;` 和语句 `i3=i1+r3;` 使 “+” 运算符产生了二义性，编译器将不知道是使用 `integer operator+(integer num1,integer num2)` 还是使用 `real operator+(real num1,real num2)`。或者说，编译器不知道在这两条语句中，是应该使用转换运算符 `integer::operator real()` 还是使用 `real::operator integer()`，因此产生二义性。必须使用显式的类型转换方式来消除二义性。例如：

```

r3=real(i1)+r1;
i3=i1+(integer)r3;

```

语句 `i3=i1-r3;` 中，因为 “-” 运算符被定义为一个成员函数，它只能被解释为 `i1.operator-(r1)`，所以不具有二义性。

【例 9-24】 将例 9-23 改为友元类的情况。

```

#include <iostream.h>

class real;

class integer
{
    long lval;
    friend real;

public:
    integer( )
    { lval=0; }
    integer(long d)
    { lval=d; }
    operator real( );
    friend integer operator+(integer num1,integer num2);
    integer operator-(integer num1);
    friend ostream & operator<<(ostream &s,integer num);

```



```
};  
class real  
{  
    float rval;  
    friend integer;  
public:  
    real() { rval=0; };  
    real(float d) { rval=d; };  
    operator integer();  
    friend real operator+(real num1,real num2);  
    real operator-(real num1);  
    friend ostream & operator<<(ostream &s,real num);  
};  
integer::operator real()  
{  
    real num;  
    num.rval=(float) lval;  
    return num;  
}  
integer operator+(integer num1,integer num2)  
{  
    integer num;  
    num.lval=num1.lval +num2.lval;  
    return num;  
}  
integer integer::operator-(integer num1)  
{  
    integer num;  
    num.lval=lval-num1.lval;  
    return num;  
}  
ostream & operator<<(ostream &s,integer num)  
{  
    return (s<<num.lval);  
}  
real::operator integer()  
{  
    integer n;  
    n.lval=(long) rval;
```

```

        return n;
    }
    real operator+(real num1,real num2)

    {
        real num;
        num.rval=num1.rval+num2.rval;
        return num;
    }
    real real::operator-(real num1)
    {
        real num;
        num.rval=rval-num1.rval;
        return num;
    }
    ostream & operator<<(ostream &s,real num)
    {
        return (s<<num.rval);
    }
    void main( )
    {
        integer i1;
        i1=50;
        cout<<"\n i1="<<i1;
        real r1=56.6;
        cout<<"\n r1="<<r1;
        integer i2;
        i2=r1;
        cout<<"\n i2=r1,i2="<<i2;
        real r2;
        r2=i1;
        cout<<"\n r2=r1,r2="<<r2;
        real r3;
        r3=real(i1)+r1;
        cout<<"\n r3=real(i1)+r1,r3="<<r3;
        r3=i1-r1;
        cout<<"\n r3=i1-r1,r3="<<r3;
        integer i3;
        i3=i1+integer(r3);
        cout<<"\n i3=i1+integer(r3),i3="<<i3;
    }

```



```
i3=i1-r3;
cout<<"\n i3=i1-r3,i3="<<i3;
integer i4=24.5;
cout<<"\n i4=24.5,i4="<<i4;
real r4=40;
cout<<"\n r4=40,r4="<<r4;
cout<<"\n";
}
```

该程序的输出仍然为：

```
i1=50
r1=56.6
i2=r1,i2=56
r2=r1,r2=50
r3=real(i1)+r1,r3=106.6
r3=i1-r1,r3=-6
i3=i1+integer(r3),i3=44
i3=i1-r3,i3=56
i4=24.5,i4=24
r4=40,r4=40
```

例 9-25 实现一个 Point 类(数据成员为一个点在两维直角坐标系内的坐标)和一个 Vector 类(数据成员为一个点在两维极坐标系内的坐标)，两个类的对象能互相赋值。

【例 9-25】

```
#include <iostream.h>
#include <math.h>
const double PI=3.14;
class Vector;
class Point
{
    friend Vector;
    int x;
    int y;
public:
    Point(int initx=0,int inity=0)
    { x=initx;y=inity;}
    operator Vector( );
    friend ostream & operator <<(ostream & stream,Point obj)
    {
```



```

        stream<<obj.x<<","<<obj.y<<"\n";
        return stream;
    }
};

class Vector
{
    friend Point ;
    double p;
    double seta;
public:
    Vector(double initp=0,double initseta=0)
    {p=initp;seta=initseta;}
    operator Point( );
    friend ostream & operator <<(ostream & stream,Vector obj)
    {
        stream<<obj.p<<","<<obj.seta<<"\n";
        return stream;
    }
};

Point::operator Vector( )
{
    Vector Vobj;
    Vobj.p=sqrt(x*x+y*y);
    if (x==0)
    {
        if (y>0) Vobj.seta=PI/2;
        else
            if (y<0 ) Vobj.seta=3*PI/2;
            else Vobj.seta=0;
    }
    else
        Vobj.seta=atan(y/x);
    return Vobj;
}

Vector::operator Point( )
{
    Point Pobj;
    Pobj.x=p*cos(seta);

```



```
Pobj.y=p*sin(seta);
return Pobj;
}
void main( )
{
    Point Pointobj1(1,2);
    Vector Vectorobj1;
    Vectorobj1=Pointobj1;
    cout<<Vectorobj1;
    Point Pointobj2;
    Vector Vectorobj2(12,3.14/3);
    Pointobj2=Vectorobj2;
    cout<<Pointobj2;
}
输出为:
2.23607,1.10715
6,10
```

关于用户定义类型转换，再强调以下两点。

① 编译器在进行类型转换时，总试图使用用户定义的类型转换函数进行类型转换。如果这样不能成功，则使用标准的类型转换机制。如果都不成功，则转换失败。

② 在类型转换具有二义性的情况下，必须使用显式类型转换，即 (type) obj 或 type (obj)，将对象 obj 转换为具有 type 类型的一个对象。

总之，类类型的对象，需要使用某个运算符，可以重载该运算，或者将类类型转换为可以使用该运算符的类型。



9.3 实例——复数类重载运算符

在前面复数类例子的基础上完善其功能，要求如下：

- ① 具有+，-，×，++，—运算功能；
- ② 能够和整数、实数进行相互转换；
- ③ 直接对复数对象完成输入/输出操作。

例 9-26 实现上述功能，其中 Complex.h 文件是复数类的头文件，Complex.cpp 文件是复数类的定义文件，Complex_main.cpp 文件测试复数类的上述功能。

【例 9-26】

① Complex.h 文件

```
#include <iostream.h>
class Complex
{
```

public:

```

    Complex(double r=0,double i=0);
    Complex(Complex & c);
    Complex &operator++();
    Complex operator++(int);
    Complex &operator--();
    Complex operator--(int);
    friend const Complex operator+(const Complex & c1,const Complex &c2);
    friend const Complex operator-(const Complex & c1,const Complex &c2);
    friend Complex operator*(const Complex & c1,const Complex &c2);
    friend istream & operator>>(istream & is, Complex &c);
    friend ostream & operator<<(ostream & os, const Complex &c);
    operator int();
    operator double();

```

private:

```

    double re,im;

```

```

};

```

② Complex.cpp 文件

```

#include <iostream.h>
#include "Complex.h"
Complex::Complex(double r,double i)
{
    re=r;
    im=i;
}
Complex::Complex(Complex & c)
{
    re=c.re;
    im=c.im;
}
Complex & Complex::operator++()
{
    re++;
    im++;
    return (*this);
}
Complex Complex::operator++(int x)
{
    Complex t(*this);

```



```
        re++;
        im++;
        return t;
    }
Complex & Complex::operator--()
{
    re--;
    im--;
    return (*this);
}
Complex Complex::operator--(int x)
{
    Complex t(*this);
    re--;
    im--;
    return t;
}
Complex::operator int()
{
    return re;
}
Complex::operator double()
{
    return re;
}
const Complex operator+(const Complex & c1,const Complex &c2)
{
    Complex t;
    t.re=c1.re+c2.re;
    t.im=c1.im+c2.im;
    return t;
}
const Complex operator-(const Complex & c1,const Complex &c2)
{
    Complex t;
    t.re=c1.re-c2.re;
    t.im=c1.im-c2.im;
    return t;
}
```

```

}
Complex operator*(const Complex & c1,const Complex &c2)
{
    Complex t;
    t.re=c1.re*c2.re-c1.im*c2.im;
    t.im=c1.re*c2.im+c1.im*c2.re;
    return t;
}
istream & operator>>(istream & is, Complex &c)
{
    is>>c.re>>c.im;
    return is;
}
ostream & operator<<(ostream & os, const Complex &c)
{
    if(c.im>0){os<<c.re<<"+"<<c.im<<"i"<<endl;}
    else {os<<c.re<<c.im<<"i"<<endl;}
    return os;
}

```

③ Complex_main.cpp 文件

```

#include "Complex.h"
int main()
{
    double re,im;
    cout<<"input the two parts of the complex:"<<endl;
    cin>>re>>im;
    Complex c1(re,im);
    cout<<"c1="<<c1;
    Complex c2(c1);
    cout<<"c2="<<c2;
    Complex c3=c1++;
    cout<<"after c3=c1++:"<<endl;
    cout<<"c1="<<c1<<"and c3="<<c3;
    Complex c4=++c1;
    cout<<"after c4=++c1:"<<endl;
    cout<<"c1="<<c1<<"and c4="<<c4;
    Complex c5=c1*c2;
    cout<<"after c5=c1*c2:"<<endl;
}

```



```
cout<<"("<<c1<<")"<<"*"<<"("<<c2<<")"<<"="<<c5<<endl;
re=c5;
cout<<"re=c5:"<<re<<endl;
int i=c5;
cout<<"i=c5:"<<i<<endl;
return 0;
}
```

运行程序，提示如下：

input the two parts of the complex:

输入：

4.5

7.2

程序运行结果如下：

```
c1=4.5+7.2i
c2=4.5+7.2i
after c3=c1++:
c1=5.5+8.2i
and c3=4.5+7.2i
after c4=++c1:
c1=6.5+9.2i
and c4=6.5+9.2i
after c5=c1*c2:
(6.5+9.2i)*(4.5+7.2i)=-36.99+88.2i
re=c5:-36.99
i=c5:-36
```

习题 9

9.1 编写一个计数器类 CCounter，要求如下：

- (1) 具有自增、自减功能；
- (2) 如果有运算符重载，请分别用成员函数和友元函数重载。

9.2 定义一个数组类，要求数组的大小在定义时初始化，而且其大小在运行时可以改变（扩充或者缩小数组空间大小），可以通过“[]”引用数组元素。测试数组功能。

9.3 C 语言和 C++ 语言中没有集合类型，可以定义一个集合类来实现。

与集合相关的操作有：

- ◎ 加入一个元素到集合中；
- ◎ 判断一个元素是否在集合中；
- ◎ 求两个集合的交集；
- ◎ 求两个集合的并集；

- ◎ 扩充集合;
- ◎ 删除集合中的一个元素;
- ◎ 输出集合的所有元素;
- ◎ 集合可以互相复制;
- ◎ 清空一个集合等。

9.4 定义类型 `vec4` 为具有 4 个 `float` 数的向量。为 `vec4` 重载运算符“[]”，使其检查参数的合法性，并为向量与浮点数的各种组合定义运算符`+`、`-`、`*`、`/`、`=`、`+=`、`-=`、`*=`和`/=`。

9.5 定义类 `mat4` 为具有 4 个 `vec4` 对象的向量，为 `mat4` 重载运算符“[]”，它返回 `vec4` 向量。为类 `mat4` 定义通常的矩阵运算。

9.6 定义一个类似于 `vec4` 的类 `vector`，但 `vec4` 的大小由构造函数 `vector::vector(int)` 的参数给定。

9.7 定义类 `matrix`，它类似于 `mat4`，但其维数由构造函数 `matrix::matrix(int,int)` 的参数给定。

第 10 章 C++语言实现继承性

继承是面向对象语言的另一个重要的概念。在客观世界中，存在着整体和部分的关系（a part of）、一般和特殊的关系（is a 或 a kind of）。

继承实现了一般和特殊的关系，解决了软件的重用性和扩充性问题。



10.1 继承和派生

系统不会凭空产生，新的软件总是在原先开发的基础上进行扩充。类确实提供了良好的模块分解技术，也具有可重用软件所期望的品质：它们是相似一致的模块，通过信息隐藏，将它们的界面与实现清楚地分开。但是，仅仅只有类是不够的，人们更希望在类的基础上取得可重用性和可扩充性的目标。

要设计可重用性模块，任何方法都必须面对重复和差别，为了避免一再地重写同样的代码，浪费时间，引入不一致的错误，必须进行抽象，寻找存在于一组类似的结构中的普遍性，例如，存在于所有表的实现中的一般特性。抽象出一般特性后，还需要在此基础上扩充其特殊的功能，使之能表达具体的东西。例如，具体的 Hash 表、数组，甚至你能想象出的任何表。因此，需要抽象技术，它能帮助产生可重用模块，也需要特殊化技术，它能在抽象的基础上，表达出具体的概念，还能帮助扩充系统。继承性提供了这些支持。



10.1.1 为什么要使用继承

自然界中，分类无处不在。如图 10-1 所示的交通工具分类层次图和图 10-2 所示的动物分类层次图，最高层次是抽象程度最高的，是最具有普遍和一般意义的概念，下层具有了上层的特性，同时加入了自己的新特性，而最下层是最具体的。在这些层次关系中，由上到下，是一个具体化、特殊化的过程；由下到上，是一个抽象的过程。

如何用程序设计语言来描述这种分类关系？类的继承和派生的层次结构，就是用来设计和描述这种自然界的分类关系的。

在最简单的情况下，类 B 继承类 A，或者从类 A 派生类 B，通常将类 A 称为基类（父类），类 B 称为派生类（子类）。这时，类 B 的对象具有类 A 对象的所有特性，甚至还会更多一些。也可以这样说，类 B 从类 A 派生出来。这意味着，类 B 至少描述了与类 A 同样的接口，至少包含了同类 A 一样的数据，可以共享类 A 的成员函数。例 10-1 说明了类 A 与类 B 之间的这种关系。

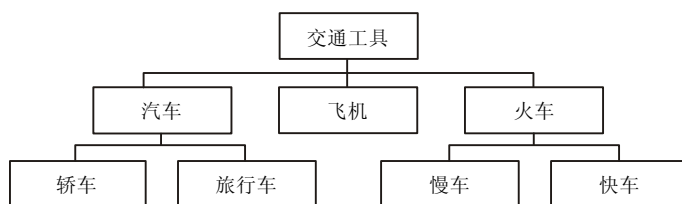


图 10-1 交通工具分类层次图

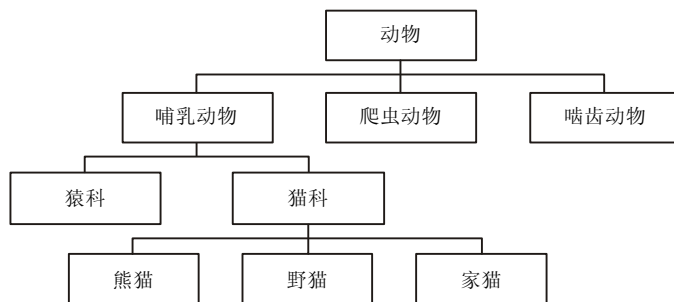


图 10-2 动物分类层次图

简单的继承的例子参见例 10-1。

【例 10-1】

```

#include<iostream.h>
class A
{
public:
    int pubA;
    void set_priA(int a)
    { priA=a;}
    void set_pubA(int a)
    { pubA=a;}
    void out_priA()
    {
        cout<<priA<<" ";
    }
private:
    int priA;
};
class B : public A
{
public:
    int pubB;

```



```
void set_priB(int a)
{ priB=a;}
void set_pubB(int a)
{ pubB=a;}
void out_B()
{ cout<<pubA<<" "<<priB<<" "<<pubB<<" "<<endl;}
private:
    int priB;
};
void main()
{
    A objA;
    objA.set_priA(1);
    objA.set_pubA(2);
    objA.out_priA();
    B objB;
    objB.set_priA(3);
    objB.set_pubA(4);
    objB.set_priB(5);
    objB.set_pubB(6);
    objB.out_priA();
    objB.out_B();
}
```

程序输出为:

1 3 4 5 6

其中, `class B: public A` 表示派生类 B 以公有派生方式继承了基类 A 的成员。通过例 10-1 可以看到, 派生类 B 除了可以调用自己的成员外, 还可以调用基类 A 的公有成员。图 10-3 显示了基类 A 拥有的成员和派生类 B 拥有的成员的关系, 其中的虚线表示存在但不能访问的基类 A 的私有成员。

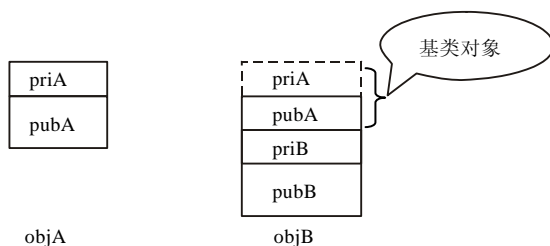


图 10-3 派生类对象包含有基类对象

字符串基本功能的实现参见例 10-2。

【例 10-2】设计一个串，满足串的基本功能，如：找出串的长度，设置串的内容，得到某个串的内容。

```
class string
{
public :
    string();
    ~string();
    int  get_length(){ return length; }
    char *  get_contents() { return  contents; }
    void  set_contents(int in_length,  char * in_contents);
    void  set_contents( char *  in_contents);

private:
    int length;
    char * contents;
};
```

这个类创建了串的基本功能接口，然而，用这些接口来处理串，功能是很有限制的。如果想对串做一些微小的修改，例如，在某一缓冲内修改串的内容，或重新设置整个串的内容，这完全能办到，但做起来不是很直截了当。

对于串增加一些编辑功能：添加、替换、删除一段正文等。但是，用具有这些特性的类来代替简单的 `string` 类仍然要包含串的基本功能，因此在新类中除了需要具有编辑功能函数外，也需要 `string` 类的基本串功能函数，但重写一来增加工作量，二来增加出错的概率，因此在已经完成的 `string` 类的基础上，扩充这些特殊的功能是一个较好的选择方案：使用继承，在 `string` 类的基础上扩充一个新类，称为 `edit_string` 类，它非常类似 `string` 类，但功能稍多一些，说明为：

```
class edit_string : public string{ };
```

这里说明的新类 `edit_string` 称为基类 `string` 的派生类。`edit_string` 类继承了基类 `string` 的所有特性，还扩充了自己的特殊功能。

基类 `string` 前面的关键字 `public` 称为访问描述符，规定了继承的一种方式：公有派生。它的含义是，在基类 `string` 中属于公有段的数据和操作，在派生类 `edit_string` 中看到的也是公有的。因此，`string` 对象有权使用的公有成员函数或公有数据，`edit_string` 对象也能使用。这时，`edit_string` 类非常类似它的基类 `string`。

除此之外，`edit_string` 类还具有自己特殊的功能。例如，用一个光标来指出处理点位置，将光标作为 `edit_string` 类的私有数据成员，说明为：

```
class edit_string : public string
{
public:
    ...

private:
```



```
int cursor;           //光标的  $x$  坐标  
};
```

如图 10-4 所示，这时的 `edit_string` 类同时包含了基类数据成员和自己特有的数据成员。

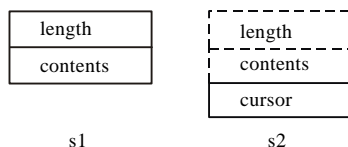


图 10-4 派生类对象包含有基类对象

然后再扩充一些操纵光标的成员函数：允许添加、删除和替换正文段的成员函数。得到如下说明（这里仅给出框架性的说明）：

```
class edit_string : public string  
{  
public :  
    int  get_cursor_pos(){ return cursor; }  
    void  move_cursor( int how_much);      // how_much 可以是正数或者负数  
    int  add_at_cursor( string * new_text );    // 返回新的 cursor  
    int  replace_at_cursor(string * new_text ); // 返回新的 cursor  
    int  delete_at_cursor( int how_much);  
private :  
    int  cursor;  
};
```

`edit_string` 类非常类似它的基类 `string` 类，并且具有移动光标，允许添加、删除和替换正文段等编辑功能，因此，`edit_string` 类是 `string` 类的扩充。

下述程序段说明了 `string`、`edit_string` 类对象的使用。

```
void main()  
{  
    string s1;  
    edit_string s2;  
    char * cp;  
    s1.set_contents(8, "get this");  
    cp = s1.get_contents( );  
    s2.set_contents(8, cp);  
    s2.move_cursor(4);  
    s1.set_contents(7, "wombats");  
    s2.replace_at_cursor(&s1);  
    ...  
}
```

主程序创建了两个对象 `s1`（`string` 类对象）和 `s2`（`edit_string` 类对象）。尽管 `edit_string`

类没有包含像 `get_contents()` 这样的成员函数说明，对象 `s2` 仍可以调用基类的成员函数 (`s2.set_contents(8, cp)`)，使用了继承，`edit_string` 类的任何对象也是 `string` 类的对象。

因此，继承就是从先辈处得到属性和行为特征。类的继承，是新的类从已有的类获取已有的特征，换个说法，从已有类产生新类的过程就是派生。类的派生实际是通过扩展、更改和特殊化，从一个已知类建立一个新类的过程。通过类的派生，可以建立具有共同关键特征的对象家族，从而实现父类代码的重用，这种继承和派生机制对于已有程序的发展和改进是极为有力的。可以描述基类和派生类的关系为：派生类是基类的具体化，基类是派生类的抽象。即基类综合了派生类的公共特征，派生类则在基类的基础上增加了某些特征，把抽象类变成具体的、实用的类型。

那么，派生类定义的成员能否与基类的成员同名呢？C++语言允许派生类可以重新定义基类的成员。如果派生类定义了与基类同名的成员，则称派生类的成员覆盖了基类的同名成员。如果要在派生类中使用基类的同名成员，可以显式地使用下述类名限定符：

类名::成员

来使用基类的成员。

例 10-3 中，派生类定义了一个数据成员 `b`，与基类的数据成员 `b` 同名，主函数 `main()` 给出了在派生类中怎样使用这两个同名成员的方法。

【例 10-3】在派生类中使用两个同名成员。

```
class base
{
public:
    int a, b;
};
class derived: public base
{
public:
    int b, c;
};
void main()
{
    derived d;
    d.a=1;
    d.base::b=2;           // 注意，base::b 使用的是 base 类的数据成员 b
    d.b=3;                 // 这里使用的是 derived 类的数据成员 b
    d.c=4;
    base * bp=&d;           // 指向基类对象的指针变量可以指向派生类的对象
                           // 因为派生类对象包含有一个基类对象
    bp->c=10;               // 错误：bp 仅能访问派生类中继承的基类成员
    bp->a=11;
```



```
        bp->b=12;  
    }
```

一个派生类从外部看，非常类似一个基类，但内部则不同。特别地，一个派生类没有权力访问它的基类的私有数据，就像其他任何类一样。例如，`edit_string` 类不能直接访问 `string` 类的 `contents` 域或 `length` 域。如果 `edit_string` 类的成员函数要使用 `contents`，则必须调用 `get_contents()` 来得到它。起初一看来，这似乎太严格了，然而，这样做的确有意义，一个类的私有成员对其他类（包括派生类）不可访问是为了确保没有其他类的成员函数依赖它们。这就允许软件适应未来的改变，赋予面向对象方法以更大的能力。

因此，派生类的生成过程包括以下 3 步。

① 继承基类的所有成员：将基类成员全盘吸收。这样，派生类实际上包含了它所有基类中除了构造和析构函数之外的所有成员。

② 改造基类成员：一种方法是通过不同的派生方式改造基类成员的访问控制问题；另一种方法是在派生类中声明一个与基类成员同名的成员，覆盖基类成员进行改造。

③ 增加新的成员：根据派生类的实际特征，增加不同于基类的成员（成员数据和成员函数）。

从编码的角度，派生类从基类中以较低的代价换来了较大的灵活性。一旦产生了可靠的基类，只需要调试派生类中所做的修改即可。派生类从基类继承属性时，可使派生类扩展它们，或者对它们进行限制，也可以改变或删除它们。所有这些变化归结为两类基本的面向对象的继承技术：性质约束和性质扩展。

这里给出了派生类的概貌，以下各节进一步考察派生类的细节。

10.1.2 派生类的声明和继承方式

为说明派生关系，派生类的说明为：

```
class edit_string : public string{...};
```

基类类名 `string` 前面的关键字 `public` 称为访问描述符。C++ 语言中，派生类的一般语法格式为：

```
class 派生类名:    <access> 基类类名, <access> 基类类名, ...  
{  
    ...  
};
```

其中，“:”后面罗列的是基类类名表，它是由“,”隔开的一串元素，每个元素都是“<access> 基类类名”。<access> 表示访问描述符，称为派生方式。派生方式有 3 种：私有派生（当 <access> 为 `private` 或默认时）、公有派生（当 <access> 为 `public` 时）和保护派生（当 <access> 为 `protect` 时）。

注意：空声明的类不能作为一个类的基类。

1. 公有派生

在公有派生情况下，基类保护成员和公有成员的访问权限在派生类中保持不变，即

- ◎ 基类的保护成员在派生类中仍然是保护成员；
- ◎ 基类的公有成员在派生类中仍然是公有成员；
- ◎ 基类的私有成员在派生类中成为派生类的不可访问成员。

一个类的不可访问成员是指该类包含有这些成员，但在类内和类外，都不能被访问。因此，对于一个派生类 A，类的成员有 4 种：私有成员、保护成员、公有成员和不可访问成员。如果该派生类还以公有方式继续派生出它的派生类 B，那么：

- ◎ A 类的保护成员在派生类 B 中仍然是保护成员；
- ◎ A 类的公有成员在派生类 B 中仍然是公有成员；
- ◎ A 类的私有成员和不可访问成员在派生类 B 中成为派生类的不可访问成员。

多层次公有派生的例子参见例 10-4。

【例 10-4】

```
#include <iostream.h>

class X {
public:
    void get_ij();
    void put_ij();
protected:
    int i,j;
};

class Y : public X
{
public:
    int get_k();
    void make_k();
private:
    int k;
};

class Z : public Y
{
public:
    void f();
};
```

这里，类 Y 由基类 X 公有派生出来，则 X 的公有段和保护段的成员在 Y 中也是公有段和保护段的成员。同时，Y 又公有派生出类 Z，这时，Y 的公有段和保护段的成员（包括从 X 继承过来的成员）在 Z 中也是公有段和保护段的成员。下面是各成员函数的实现代码。

```
void X::get_ij()
{    cout << "Enter the two numbers";    cin >> i >> j; }

void X::put_ij()
```



```
{    cout << i << " "<< j << "\n"; }
int Y::get_k()
{    return k; }
void Y::make_k()
{    k=i*j; }
void Z::f()
{    i=2;    j=3; }
```

可见，在 X 和 Z 中都可以直接使用基类 X 的保护段中的成员 i 和 j。下面的 main 函数说明了怎样使用这些派生类。

```
void main()
{
    Y objY;
    Z objZ;
    objY.get_ij();
    objY.put_ij();
    objY.make_k();
    cout << objY.get_k() << "\n";
    objZ.f();
    objZ.put_ij();
}
```

总之，一个派生类如果是从基类公有派生出来的，则基类成员的公有和保护成员的访问权限在派生类中保持不变。

在公有派生方式下，一个派生类的对象可以作为基类的对象使用（称为赋值兼容规则），具体有以下 3 种情况：

- ① 派生类的对象可以直接赋给基类的对象；
- ② 基类对象的引用可以引用一个派生类对象；
- ③ 基类对象的指针可以指向一个派生类对象。

设基类为 Base，公有派生类为 Derived，代码如下：

```
Base  Bobj;
Derived Dobj;
...
Bobj=Dobj;                //正确
Base &Bref=Dobj;          //正确
Base * Bpointer=&Dobj;     //正确
```

也就是说，当基类指针指向派生对象时，实际指向的是派生对象拥有的基类成员，因此，可以访问基类的公有成员，而不能访问派生类的成员，同时基类在派生类中的不可访问成员此时仍然不能通过基类指针访问。

2. 私有派生

如果〈access〉为 `private`（或者默认），那么基类的所有公有段和保护段的成员都成为派生类私有段的成员，称派生类是基类的私有派生类。

- ◎ 基类的保护成员在私有派生类中是私有成员；
- ◎ 基类的公有成员在私有派生类中是私有成员；
- ◎ 基类的私有成员和不可访问成员在私有派生类中成为派生类的不可访问成员。

也就是说，私有派生之后，基类的成员就再也无法在以后的派生类中发挥作用，相当于阻止了基类功能的继续派生。因此，在一般情况下，私有派生使用比较少。

私有派生的例子参见例 10-5。

【例 10-5】

```
#include <iostream.h>

class X
{
public:
    void get_ij();
    void put_ij();
protected:
    int i,j;
};

class Y : private X
{
public:
    int get_k();
    void make_k();
private:
    int k;
};

class Z : public Y
{
public:
    void f();
};
```

这里，派生类 `Y` 由基类 `X` 私有派生，`X` 的公有段和保护段的成员在 `Y` 中变成了私有段的成员。同时，`Z` 又由类 `Y` 公有派生，则 `Y` 的公有段和保护段的成员（并不包括从 `X` 继承过来的成员，它们属于 `Y` 的私有段）成为 `Z` 的公有段和保护段成员。下面是各成员函数的实现代码。

```
void X::get_ij()
{
```



```
        cout << "Enter the two numbers";
        cin >> i >> j;
    }
    void X::put_ij()
    {
        cout << i << " "<< j << "\n";
    }
    int Y::get_k()
    {
        return k;
    }
    void Y::make_k()
    {
        k=i*j;
    }
    void Z::f()
    {
        i=2;           //错误
        j=3;           //错误
    }
```

类 X 的保护段成员 i 和 j 已经成了类 Y 的私有段成员，因此 Y 的派生类 Z 及 Z 的对象都不能访问 X 的保护段成员 i 和 j，以及公有段成员函数 get_ij()和 put_ij()。下面的 main 函数说明了这些派生类的使用。

```
void main()
{
    Y objY;
    Z objZ;
    objY.get_ij();           //错误
    objY.put_ij();           //错误  objY.make_k();
    cout << objY.get_k() << "\n";
    objZ.put_ij();           //错误
}
```

如果希望私有派生的基类的部分成员能够被类外或者再次派生的类使用，可以使用覆盖的方法，这是对原有成员改造的重要手段，是程序设计的常用方法。

例 10-5 的公有段成员函数 get_ij()如果希望能够被访问，则修改类 Y 接口如下：

```
class Y : private X
{
public :
```

```

    int get_k();
    void make_k();
    void get_ij(){return X::get_ij();}
private:
    int k;
};

```

下面的程序段隐含了一个由私有派生引起的错误，请读者指出 `void g(Y * P)` 函数体中错误的原因。

```

class X
{
public:
    void f( );
};
class Y: X
{ };
void g(Y * p)
{
    p->f();           //错误
}

```

3. 保护派生

(1) 保护成员

每个人都有秘密，有的秘密会随着人的离去而成为永远的秘密，有的秘密却会在人即将逝去的时候告诉自己最信任和亲密的人，比如子女。那么在 C++ 程序设计语言中，如何表示这种关系呢？

如果用私有类型表示自己永远的秘密，那么，那些希望只能被派生类访问，但不能被其他类访问的这些秘密如何表示呢？我们知道，一个派生类继承其基类时，基类的所有私有段的成员不能被派生类访问，参见例 10-6。

【例 10-6】

```

class Father
{
public:
    void buy_house();
    void sell_house();
private:
    int house;
    char * lover;
    int money;
};

```



```
class Son : public Father
{
public:
    void buy_stock(int s);
    int sell_stock(int s);
private:
    int stock;
};
int Son::sell_stock(int s)
{
    if(stock<s)
    {
        stock=stock+money;           // 错误
        if(stock<s)
        {
            cout<<"not enough money "<<endl;
            return 0;
        }
    }
    stock=stock-s;
    return 1;
}
```

例 10-6 中，该成员函数是不能工作的，因为 `money` 是基类 `Father` 的私有成员，不能被派生类 `Son` 访问。如果 `Father` 同意 `Son` 使用他的钱，但其他人不能使用，怎么办？即基类是否存在这样一类成员，它们可以被派生类访问，但不能被其他类访问。也就是说，除了基类和它的派生类可以看到这部分成员外，对其他类如同私有段成员一样。保护段成员正是为了满足这一要求而提出的。

重新定义 `Father` 类为：

```
class Father
{
public:
    void buy_house();
    void sell_house();
private:
    int house;
    char * lover;
protected:
    int money;
```

```

};
int Son::sell_stock(int s)
{
    if(stock<s)
    {
        stock=stock+money;
        if(stock<s)
        {
            cout<<"not enough money "<<endl;
            return 0;
        }
    }
    stock=stock-s;
    return 1;
}

```

这样就能正确工作了。但是，在类 **Father** 和类 **Son** 的外部，人们仍然不能访问 **money**。除了基类和它的派生类外，保护段成员同私有段成员一样被隐藏起来。如果合理利用保护成员，就可以在类的复杂层次关系中找到一个平衡点，既能够实现成员隐蔽，又能够方便继承，实现代码的高效重用和扩充。

总之，类中可以有 3 种成员：

```

class 类名
{
    private:
        ... //私有成员段在类外不能被访问（除非友元关系）；
           //私有成员段不能被继承
    protected:
        ... //保护段成员在类外不能被访问（除非友元关系）；
           //保护段成员能被继承
    public:
        ... //公有段成员在类外能被访问；
           //公有段成员能被继承
};

```

（2）保护派生

```
class Derived: protected Base
```

C++语言还允许保护派生方式。在保护派生方式下，基类的所有公有段成员和保护段成员都成为保护派生类保护段的成员，基类的私有成员和不可访问成员在保护派生类中成为派生类的不可访问成员。

保护派生方式一般很少使用。



4. 静态成员的派生

static 成员受段约束符的限制,但不受访问描述符的限制。基类和派生类共享基类的 static 成员。

不管公有派生还是私有派生都不影响派生类对基类的静态成员的访问,但要求访问静态成员时,必须用“类名::成员”形式显式地说明,参见例 10-7。

【例 10-7】

```
#include <iostream.h>
class S
{
public:
    static void set_s1()
    {
        s1=1;
    }
    static void out_s1()
    {
        cout<<s1;
    }
    static void set_s2()
    {
        s2=1;
    }
    static void out_s2()
    {
        cout<<s2;
    }
    static void set_s3()
    {
        s3=1;
    }
    static void out_s3()
    {
        cout<<s3;
    }
    static int s1;
protected:
    static int s2;
private:
```

```

        static int s3;
    };
    int S::s1=0;
    int S::s2=0;
    int S::s3=0;
    class D : private S
    {
    public:
        void set()
        {
            S::s1=10;
            S::s2=20;           //错误
            S::s3=30;           //错误
        }
    };
    void main()
    {
        S objS;
        S::s1=0;
        S::s2=0;               //错误：类外访问保护成员
        S::s3=0;               //错误：类外访问私有成员
        objS.set_s1();
        objS.set_s2();
        objS.set_s3();
        objS.out_s1();
        D objD;
        objD.set();
        objD.out_s1();         //错误，不能访问 S 类的公有成员（私有派生）
    }

```

可见，派生对于基类的 static 成员是没有影响的。但是类外只能访问公有成员，不能访问私有和保护类成员。

10.1.3 基类对象的初始化

1. 类等级

如图 10-1 所示，类交通工具被称为类汽车的直接基类。类汽车也是类轿车的直接基类。类交通工具不是类轿车的直接基类，称类交通工具为类轿车的间接基类。这样，类交通工具、汽车和轿车形成了一个类等级。



在类等级中，若一个类只有一个直接基类，则这种继承关系称为单继承。若一个类有多个直接基类，则这种继承关系称为多继承。

在一个派生类中要访问与派生类成员同名的基类成员，使用“类名::成员”的方式，其中类名可以是某个间接基类的类名，因而成员也是这个间接基类的成员，参见例 10-8。

【例 10-8】

```
class A
{
public:
    void f();
    ...
};
class B: public A
{
    ...
};
class C: public B
{
public:
    void f();
    void g();
};
void C::g()
{
    f();           //调用 C 类 的 f()
    A::f();        //调用 A 的 f()
    B::f();        //调用 B 的 f(), 也就是 从类 A 继承的 f()
}
void main( )
{
    C Cobj;
    Cobj.f();      //调用 C 类的 f()
    Cobj.A::f();   //调用 A 的 f()
    Cobj.B::f();   //调用 B 的 f(), 也就是从类 A 继承的 f()
    ...
}
```

若类 B 中也定义了自己的同名函数 f(); 则 B::f();语句调用的就是类 B 的 f()。

2. 基类对象的初始化

前面讨论了派生类的派生过程，首先是继承基类的成员，再修改和增加派生类具有自己

特性的成员。但是基类的构造函数、析构函数和 `operator=` 都不能被派生类继承。在建立一个类等级后, 通常是通过创建某个派生类的对象来使用这个类等级, 包括隐含地使用基类的数据和函数, 派生类对象创建的时候, 一定会调用构造函数初始化该类对象, 且一定会先调用基类构造函数。那么当创建一个派生类对象时, 怎样调用基类的构造函数对基类数据初始化? 显然, 需要提供一种初始化机制使得派生类对象在创建时, 能够通过访问基类的构造函数来初始化基类的数据。C++语言在派生类的构造函数中提供这种初始化基类的机制。

在 C++语言中, 派生类构造函数的声明为:

派生类构造函数(变元表): 基类(变元表), 对象成员 1(变元表), ..., 对象成员 n(变元表)

```
{
    ...
}
```

这种初始化机制与 8.11.3 节中的对象成员的初始化机制是一致的。“:”后面是基类的构造函数(多继承中有多个基类的构造函数)以及对象成员的初始化表。这个构造函数执行时仍遵循先父辈(基类), 再客入(对象成员), 后自己(派生类)的顺序。如果基类使用默认构造函数或不带参数的构造函数, 那么派生类构造函数声明中“:”后面的“基类(变元表)”一项可以省去, 但是派生类构造函数执行时仍然隐含地调用基类的构造函数执行。

例 10-9 显示了构造函数的执行顺序, 先父辈, 后自己。

【例 10-9】

```
#include <iostream.h>

class Base
{
public:
    Base() {cout<<"\nBase created \n";}
};

class D_class:public Base
{
public:
    D_class() {cout<<"D_class created \n";}
};

void main()
{
    D_class d;
}
```

程序输出:

```
Base created
D_class created
```

先执行基类的构造函数, 再执行派生类的构造函数。由于基类对派生类一无所知, 虽然基类的初始化与派生类的初始化相分离, 但它可能是派生类的先决条件。因此, 初始化时,



先执行基类的初始化。

另一方面，执行析构函数时，先执行派生类的析构函数，再执行基类的析构函数。原因是可理解的，对基类的破坏隐含了对派生类的破坏，所以派生类的析构函数必须先执行。基类的构造函数先执行，这点很重要，分析例 10-10。

【例 10-10】

```
class Base
{
public:
    Base(int i):x(i){ }
private:
    int x;
};
class Derived:public Base
{
public:
    Derived(int i):a(i* 10), Base(a){ }
private:
    int a;
};
```

当执行 `Derived d(1);` 语句时，`d(1)` 将先引起基类构造函数 `Base(a)` 的调用，这时传给构造函数的值 `a` 是未定义的，而不是将初始化的 `a` 值传给基类 `Base`，因此，基类的没有用希望的值正确的初始化。例 10-11 说明了构造函数的执行情况。

【例 10-11】

```
#include <iostream.h>
#include <stdio.h>
class Parent
{
public:
    Parent( int p1, int p2 )
    {
        private1=p1;
        private2=p2;
    }
    int inc1(){ return ++private1;}
    int inc2(){ return ++private2;}
    void display()
    {
        printf("\n private1=%d,   private2=%d",private1,private2);
```

```

    }
private:
    int private1,private2;
};
class Derived:private Parent
{
public:
    Derived(int p1,int p2,int p3,int p4,int p5):Parent(p1,p2),private4(p3,p4)
    { private3=p5;}
    int inc1()
    { return Parent:: inc1();}
    int inc3()
    { return ++private3;}
    void display()
    {
        Parent::display();
        private4.Parent::display();
        printf("\n private3=%d\n", private3);
    }
private:
    int private3;
    Parent private4;
};

```

基类 Parent 有一个参数化的构造函数 Parent(int p1,int p2), 派生类 Derived 含有一个对象成员: private4。为实现初始化, 派生类构造函数声明为:

```

Derived(int p1,int p2,int p3,int p4,int p5):Parent(p1,p2),private4(p3, p4)
{
    private3=p5;
}

```

这个构造函数有 5 个参数, 前两个参数 p1、p2 用于初始化基类的私有数据 private1 和 private2。在初始化表中使用构造函数 Parent(p1,p2)进行初始化; 参数 p3、p4 用于初始化对象成员 private4; 参数 p5 用于初始化私有数据 private3。该构造函数的执行次序是: 首先基类, 其次对象成员, 最后派生类。

下面 main()函数说明怎样创建派生类对象:

```

void main()
{
    Derived d1(17,18,1,2,-5);
    d1.inc1();
}

```



```
        d1.display();  
    }
```

该程序的输出为：

```
private1=18,  private2=18  
private1=1,  private2=2  
private3=-5
```

注意：派生类的友元只能访问基类的保护段成员和公有段成员及派生类自己的所有成员。

至此，已经介绍了派生类的基本概念、派生类的语法、保护段、访问描述符和派生类的初始化，下面用例 10-12 说明继承的使用。

【例 10-12】 队列和堆栈的实现（定义一个数组类，派生出队列和堆栈）

```
#include <iostream.h>  
  
class Array  
{  
protected:  
    int * p;  
    int size;  
public:  
    Array(int num)  
    {  
        size=(num>6)?num:6;  
        p=new int[size];  
    }  
    ~Array()  
    {  
        delete [] p;  
    }  
    int & operator[](int idx)    //超载[]  
    {  
        if (idx<size)  
            return p[idx];  
        else  
        {  
            expend(idx-size+1);  
            return p[idx];  
        }  
    }  
    void expend(int offset)
```

```

    {
        int * pi;
        pi=new int [size+offset];
        for (int num=0;num<size;num++)
            pi[num]=p[num];
        delete [] p;
        p=pi;
        size=size+offset;
    }
    void contract(int offset)
    {
        size=size-offset;
    }
};

class Queue:public Array
{
    int first,last;
public :
    Queue(int a):Array(a)
    {    first=last=0;}
    void join_queue(int x)
    {
        if(last==size)
            expend(1);
        p[last++]=x;
    }
    int get()
    {
        if(first<last)
            return p[first++];
        else
        {
            cout<<"空队";
            return -1;
        }
    }
};

class Stack:public Array
{

```



```
int top;
public :
    Stack(int a):Array(a)
    {    top=0;}
    void push(int x)
    {
        if(top==size)
            expend(1);
        p[top++]=x;
    }
    int pop()
    {
        if(top>0)
            return p[--top];
        else
        {
            cout<<"空堆栈";
            return -1;
        }
    }
};
```



10.2 多继承



10.2.1 多继承的概念

至今所看到的例子中，派生类仅有一个直接基类，这称为单继承。但是一些类却代表两个或多个类的合成。例如，两用沙发，它是一张沙发，也是一张床，两用沙发同时继承沙发和床的特征，即 SleepSofa 继承 Bed 和 Sofa 两个类，因此多继承是指一个派生类有两个或者两个以上的直接基类。两用沙发类的层次结构如图 10-5 所示，其程序代码参见例 10-13。

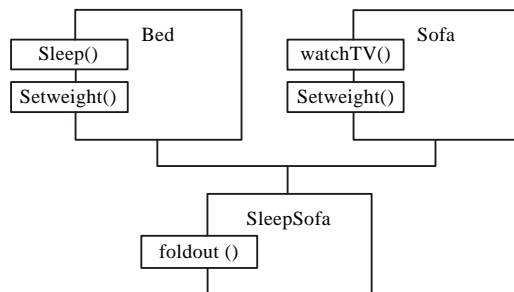


图 10-5 两用沙发的类层次结构

【例 10-13】

```

#include<iostream.h>

class Bed
{
public:
    Bed(int w):weight(w){};
    void Sleep(){ cout <<"Sleeping..."<<endl;}
    void SetWeight(int w){weight=w;}

protected:
    int weight;
};

class Sofa
{
public:
    Sofa(int w):weight(w){};
    void WatchTV(){ cout <<"Watching TV..."<<endl;};
    void SetWeight(int w){ weight=w;}

protected:
    int weight;
};

class SleepSofa:public Bed, public Sofa
{
public:
    SleepSofa(int w):Bed(w),Sofa(w){fold=1;};//沙发开始是折叠的
    void Foldout();
    int SetFold(int f){fold=f;}
    void SetWeight(int w);

protected:
    int fold;
};

void SleepSofa::Foldout()
{
    if(fold!=0)
        cout <<"Fold out the sofa"<<endl;
    else
        cout <<"Not Fold.It's a bed "<<endl;
}

void SleepSofa::SetWeight(int w)

```



```
{
    Bed::SetWeight(w);
    Sofa::SetWeight(w);
}
void main()
{
    int w;
    cout << "please input the weight of the SleepSofa: " << endl;
    cin >> w;
    SleepSofa ss(w);
    ss.WatchTV();
    ss.Foldout();
    ss.Sleep();
}
```

运行结果为：

```
please input the weight of the SleepSofa:
34
Watching TV...
Fold out the sofa
Sleeping...
```

两用沙发继承两个基类的所有成员，这样，`ss.Sleep()`和`ss.WatchTV()`的调用是合法的。也就可以把 `SleepSofa` 当做一个 `Bed` 或一个 `Sofa` 用。另外，`SleepSofa` 类还有它自己的成员 `Foldout()`，`SetFold(int f)` 和 `SetWeight(int w)`。至此，已建立了多继承的类等级。所谓多继承，就是派生类具有多个直接基类，因此，它的说明语法与单继承不同之处在于有多个基类类名，这些基类类名之间由“，”隔开。要求这些基类类名不能两两相同，即一个类不能被多次说明为一个派生类的直接基类。

`SleepSofa(int w):Bed(w),Sofa(w){fold=1;};`是一个多继承派生类的构造函数，它通过调用两个基类构造函数，对基类数据进行初始化。

构造函数体 `fold=1;`实现对成员数据的初始化，因为所有必要的基类初始化工作都已在成员初始化表中完成了。当 `SleepSofa` 构造函数被调用时，暗中将触发大量的工作。

首先，调用 `Bed` 的构造函数，然后调用 `Sofa` 的构造函数。在 `SleepSofa` 构造函数中所给定的参数被用来初始化基类中相应的数据成员，最后初始化 `SleepSofa` 类自身的数据成员。

当调用析构函数时，调用顺序与构造函数的顺序相反。如果用户没有提供自己的构造函数或析构函数，C++语言将产生和调用默认版本。

在 `SleepSofa::SetWeight(int w)` 定义中，可看到以下两个函数，类名限定了调用哪一个类的 `SetWeight` 函数：

```
void SleepSofa::SetWeight(int w)
{
```



```

        Bed::SetWeight(w);
        Sofa::SetWeight(w);
    }

```

在这个类等级中给每种类定义了一个 `SetWeight` 函数：`SleepSofa::SetWeight (int w)`、`Bed::SetWeight (int w)`和 `Sofa::SetWeight (int w)`。这些 `SetWeight` 函数都有完全相同的函数原型，调用哪一个 `SetWeight` 呢？它们的区分是在编译时进行的。实际上，到目前为止，区分同名函数有三种机制。

① 根据参数的特征加以区分，例如：

```
SetWeight (int, char)
```

与 `SetWeight (char * , float)`

不是同一个函数，编译时能根据参数类型进行区分。

② 使用 “::” 加以区分，例如：`Bed::SetWeight` 有别于 `Sofa::SetWeight` 通过限定类名，编译时能够区分。

③ 根据类对象来区分，例如：

```
ABed. SetWeight (w)    调用    Bed :: SetWeight (int w)
```

```
ASofa. SetWeight (w)   调用    Sofa :: SetWeight (int w)
```

这里，`ABed` 是 `Bed` 的对象，`ASofa` 是类 `Sofa` 的对象。通过对象，可以区分此刻调用哪一个 `SetWeight` 版本。

这里所有函数的区分都是在编译时进行的，称为早期匹配。C++语言提供了一种更灵活的机制来解决函数匹配问题，它允许 `SetWeight ()` 的使用与 `SetWeight ()` 的实现版本之间的联系推迟到运行时进行（而不是编译时确定），称为晚期匹配。虚函数正是解决这些问题的关键概念。

10.2.2 虚基类

虚基类在实际编程中是很有用的，但它的语义很难处理。这里介绍目前实现的虚基类的概念，第一次阅读时可跳过本节。

1. 虚基类的概念

在 C++语言中，一个类不能被多次说明为一个派生类的直接基类，但可以不止一次地成为间接基类。这就导致了一些问题，参见例 10-14。

【例 10-14】

```

class L
{
public:
    int next;
};
class A : public L
{

```

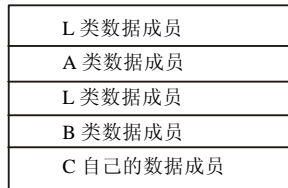
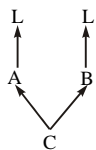


```
};  
class B : public L  
{  
};  
class C : public A, public B  
{  
public:  
    void f() { next = 0;}    //错误  
};
```

这里，类 L 两次成为类 C 的间接基类。这意味着，C 类对象中有两个 L 对象，由类 A 继承的 L 对象和由类 B 继承的 L 对象，如图 10-6 所示。

现在考察 C::f() 函数。这时，next 有两个拷贝(A 继承路径的 next 和 B 继承路径的 next)，语句 next = 0; 具有二义性，它是将 A::next 置为 0？还是将 B::next 置为 0？或者将两者都置为 0？这需要在函数 f() 中被显式地说明。

如果希望间接基类 L 与其派生类的关系是图 10-7 所示的情况，而不是图 10-6 所示的情况。显然，目前多继承的方法不能描述这种情况，需要新的描述方法，显式地指明间接基类与其派生类的这种单拷贝关系。C++ 语言提供了这种描述手段。它将 L 说明为 A 和 B 的虚基类。



C 的对象

图 10-6 类 C 的继承关系

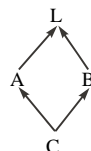


图 10-7 虚基类

当在多条继承路径上有一个公共的基类（如本例的 L）时，在这些路径中的某几条路径汇合处（如本例中的 C），这个公共基类就会产生多个实例。如果只想保存这个基类的一个实例，可以将这个公共基类说明为虚拟基类或称虚基类。它仅仅简单地将关键字 virtual 加到基类的描述上，参见例 10-15。

【例 10-15】

```
class L  
{  
public:  
    int next;  
};  
class A : virtual public L    //或 class A :public virtual L  
{
```

```

};
class B : virtual public L           //或 class B: public virtual L
{
};
class C : public A, public B
{
public:
    void f() { next = 0;}
};

```

这时 C 类对象中只有 L 的一个拷贝，因而函数 C::f() 中的语句 next=0; 没有二义性。在此例中，对于类 C 而言，L 类是 B 类的虚（假）基类，是类 A 的真基类；但对于类 B 而言，L 类还是 B 类的真基类。虚基类只是一个相对的概念，关键要考虑以什么顺序来继续向下派生类，参见例 10-16。

【例 10-16】

```

class L
{
public:
    int next;
};
class A : virtual public L
{
};
class B : virtual public L
{
};
class C : public B, public A
{
public:
    void f() { next = 0;}
};

```

在此例中，对于 C 类而言，L 类是 A 类的虚（假）基类，是类 B 的真基类。

同单继承相比，多继承的一个重要性质是，类格中的类不需把信息推向一棵继承树的单个根就可以共享这些信息。在效果上，单一继承树的根结点充当继承树中所有类的全局名，并且它承受着一种在没有继承性的语言中所见到的现象，即所有感兴趣的信息以成为全局量为终点。这时，产生信息向树的顶层漂移的倾向。虚基类提供了一种共享信息的局部点，使得局部引用和信息隐藏有所改进。由于虚基类是一种十分普通的 C++ 类，因而语言的所有特性都可用于表达这种共享。

一个派生类的对象的地址可以直接赋给虚基类的指针，例如：



```
C obj;  
L * ptr = &obj;
```

这时不需要强制类型转换。并且，一个虚基类的引用可以引用一个派生类的对象，例如：

```
C obj2;  
L &ref = obj2;
```

反之则不行。无论在强制类型转换中指定什么路径，一个虚基类的指针或引用都不能转换为派生类的指针或引用。例如：

```
C * p =(C*)(A*) ptr;
```

将产生编译错误。

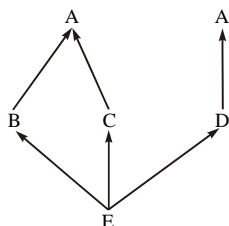


图 10-8 多重继承图

注意：为了保证虚基类在派生类中只继承一次，应当在该基类的所有直接派生类中声明为虚基类，否则，仍然会出现对基类的多次继承。如图 10-8 所示，在派生类 B 和 C 中将类 A 声明为虚基类，而在派生类 D 中没有将类 A 声明为虚基类，则在派生类 E 中，虽然从类 B 和 C 路径派生的部分只保留一份基类成员，但从类 D 路径派生的部分还保留一份基类成员。

2. 虚基类对象的初始化

在 Borland C++ 3.0 中，虚基类的初始化与多继承的初始化在语法上是一样的，但隐含的构造函数的调用顺序有点差别。

虚基类构造函数的调用顺序是这样规定的：

- ① 虚基类的构造函数在非虚基类之前调用；
- ② 若同一层次中包含多个虚基类，则虚基类构造函数按它们说明的次序调用；
- ③ 若虚基类由非虚基类派生，则遵守先调用基类构造函数，再调用派生类构造函数的规则。

例如：

```
class X : public Y, virtual public Z  
{  
}  
  
X one;
```

将产生如下的构造函数调用顺序：

```
Z()  
Y()  
X()
```

这里，Z 是 X 的虚基类，故先调用 Z 的构造函数，再调用 Y 的构造函数，最后才调用派生类 X 自己的构造函数。虚基类构造函数调用顺序参见例 10-17。

【例 10-17】

```
#include <iostream.h>  
class base
```

```

{
public:
    base(){cout<<"base()"<<endl;}
    ~base(){cout<<"~base()"<<endl;}
};
class base2
{
public:
    base2(){cout<<"base2()"<<endl;}
    ~base2(){cout<<"~base2()"<<endl;}
};
class level1 : public base2, virtual public base
{
public:
    level1(){cout<<"level1()"<<endl;}
    ~level1(){cout<<"~level1()"<<endl;}
};
class level2 : public base2, virtual public base
{
public:
    level2(){cout<<"level2()"<<endl;}
    ~level2(){cout<<"~level2()"<<endl;}
};
class toplevel : public level1, virtual public level2
{
public:
    toplevel(){cout<<"toplevel()"<<endl;}
    ~toplevel(){cout<<"~toplevel()"<<endl;}
};
void main()
{
    toplevel view;
}

```

当建立对象 view 时，调用顺序为：level2()→level1()→toplevel()。而 level2()要求的调用顺序为：base()→base2()→level2()，level1()要求的调用顺序为：base()→base2()→level1()，toplevel()要求：toplevel()。

所以，构造函数的调用顺序为：

base()



```
base2()
level2()
base2()
level1()
toplevel()
```

toplevel 有两个基类：一个是虚基类 level2，另一个是非虚基类 level1。遵循规定，应该先执行 level2 的构造函数。level2 也有两个基类，一个是虚基类 base，另一个是非虚基类

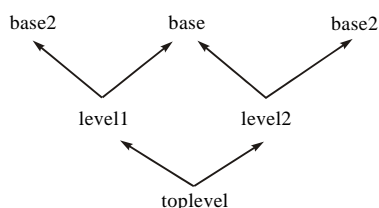


图 10-9 虚基类和非虚基类 (base 是 level1 的虚基类)

base2。应该先执行 base 的构造函数，因为 base 没有基类了，所以首先执行 base 的构造函数，再执行 base2 的构造函数，最后执行 level2 的构造函数。toplevel 还有一个基类 level1，而 level1 又有两个基类：base 是虚基类，无须再执行其构造函数；base2 是非虚基类，还要先执行 base2 的构造函数，再执行 level1 的构造函数，最后执行 toplevel 的构造函数。

在例 10-17 中，对于 toplevel 的对象而言，base 是 level1 的虚基类，类关系如图 10-9 所示。

修改派生顺序的例子参见例 10-18。

【例 10-18】

```
class base
{...};
class base2
{...};
class level1 : public base2, virtual public base
{...};
class level2 : public base2, virtual public base
{...};
class toplevel : virtual public level1, public level2
{...};
void main()
{
    toplevel view;
}
```

当建立对象 view 时，调用顺序为：

```
level1(): base()→base2()→level1();
level2(): base2()→level2();
toplevel(): toplevel()。
```

在本例中，对于 toplevel 的而言，base 是 level2 的虚基类，如图 10-10 所示。虚基类与非虚基类的区别见例 10-19。

【例 10-19】

```

#include <iostream.h>
class B
{
public:
    int b;
};
class X : virtual public B
{
};
class Y : virtual public B
{
};
class Z : public B
{
};
class AA: public X, public Y, public Z
{
    void f(){b++;};
};
void main()
{
}

```

这里，AA 具有两个 B 类的子对象：Z 的 B 和 X 与 Y 共享的虚拟的 B，如图 10-11 所示。程序编译不能通过，提示 AA::b 具有二义性。因为 AA::b 有两个：一个从 Z 的路径继承，另一个从 X 与 Y 共享的虚拟 B 继承。

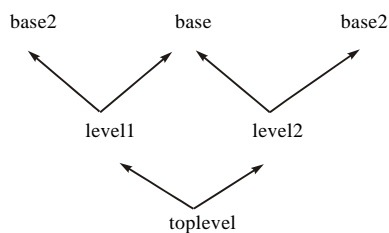


图 10-10 虚基类和非虚基类（base 是 level2 的虚基类）

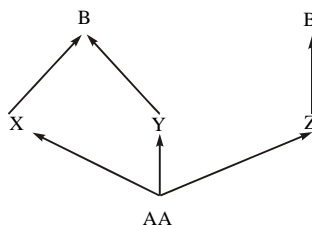


图 10-11 虚基类和非虚基类



10.3 继承的意义

继承性是对象之间合作的另一种方式（另两种方式是友元类和对象作为成员），派生类继承了基类，一个派生类对象除了可以包含基类对象（这一点和对象作为成员类似）外，派生类还可以继承基类中的成员。派生类对象可以在类外直接使用继承的基类公有成员。

类既可看做模块又可看做类型。继承的引入使这两种观点更加明显，从模块和类型两个



角度来仔细考察继承的意义。

10.3.1 模块的观点

从模块观点看，继承是一种关键的可重用和可扩充技术。

模块提供给外部世界一组服务。没有继承，每个新模块必须定义自己所有这些服务。当然，这些服务的实现可能依靠其他模块提供，但是没有什么方法来定义新模块，使得新模块只是简单地加新服务到原先已定义好的模块上。

继承提供了一种可能性，即如果类 B 从类 A 继承，A 的所有服务（特性）将自动提供给 B，没有必要进一步定义它们。B 除为了自己的目的增加新的服务（特性）外，还具有重定义基类的服务，以适应自己需要的灵活性。

这就提供了一种完全不同于传统方法的软件开发方法，不是解决每一个新课题都试图从便笺（scratch）开始，而是在预先实现的基础上来扩充其结果，不需要重做已经做过的东西。因此，一个好的模块结构应该既是开放的，又是闭合的，这就是开放闭合原理。它可陈述为：

① 一个模块如果提供了扩充的可能性，则称为模块是开放的，例如，模块提供了数据结构和过程的可能；

② 一个模块如果能交付其他模块使用，则称为模块是闭合的，例如，一个能被编译的，存储在库中提供使用的模块。

要求一个模块结构是开放的，因为很难保证在软件演化过程中不扩充这些服务；要求一个模块结构是闭合的，因为顾客要求模块服务能够稳定地使用，不要总变来变去。

这种双重的需要使人很为难，传统的模块结构基本上没有提供什么解决方案，但继承性解决了它。一个类是闭合的，因为它可以编译，存储在一个库中，提供顾客使用；同时它又是开放的，通过继承性提供了扩充的可能。当需要新服务时，只需定义一个派生类来继承基类的特性，并扩充自己特殊的需要，这对基类的顾客没有任何打扰。

设计可重用模块结构最困难的问题是寻找共同性。例如，所有的 hash 表、所有的顺序表之间的共同性。至少有一种方法可以这样做：通过使用继承来建立类等级，就可以充分抽象出各个层次的共同特性。

10.3.2 类型的观点

从类型角度，继承性涉及了可重用性和灵活性，这里的关键是动态匹配。类型是一组由操作来表征的值（从抽象数据类型的观点）。整型描述了一组具有算术操作的数；多边形描述了一组具有顶点和周长的对象。根据这种观点，继承描述了通常称为 is_a 的关系。例如，“狗是一种哺乳动物”，“哺乳动物是一种动物”等，类似地，“长方形是一种多边形”。这种关系意味着什么呢？如果考虑每一种类型的值，这个关系就是一种集包含关系。狗类是动物类的子集。尽管长方形的实例不是多边形实例的子集，但由长方形类型的函数引用的对象是由多边形类型的函数引用的对象的子集。

继承关系体现了 A is a kind of B 的关系。

如果考虑应用到每种类型的操作，类型 B 是类型 A 的子集意味着：每一个适用于 A 的实例的操作也可适用于 B 的实例（当然 B 中也可能重定义 A 中的操作）。这里重定义和动态匹配起着基本的作用。使用继承，能方便地描述 is_a 分层的关系。

如图 10-12 所示，在这个例子中，多边形（polygon 类）和长方形（rectangle 类）是两个子类，对于平移（translate）、旋转（rotate）、显示（display）等操作，每个子类都可以定义自己的版本。

关键是这种方法所允许的分布式软件结构。在这个 is_a 分层中，每一子类都对应于一种具体的操作实现。而在传统方法中，如 Pascal 或 Ada，对于 rotate 操作，可能会有如下结构：

```
case f.figuretype of
    polygon: (...);
    circle: (...);
    ...
end
```

麻烦在于，这种程序结构具有整个系统太多的知识，每个模块都必须确切知道什么样的 figure 类型在系统中是允许的，当添加一个新的图形类型或者改变存在的某个图形类型时，将影响整个程序。

这种将太多的知识分布在程序中是传统方法不灵活的主要原因，软件修改的大部分困难都来源于它，这就部分解释了为什么软件项目这样难以保持控制。一个显然很小的改变将影响许多模块，强迫开发者来重新打开已闭合得很好的模块。

用面向对象方法能补救这个问题。一个操作的特殊实现的变化将仅仅影响实现所应用的类。添加一个新类型在许多情况下不会影响其他类。再者，分布性是关键：类管理自己的事务，不干涉其他内政，各人自扫门前雪，这是获得分布式结构的要求。

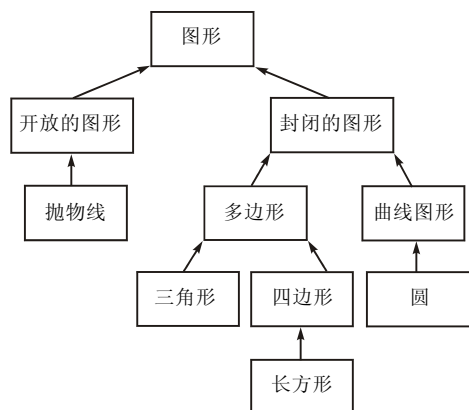


图 10-12 图形类的继承例子



10.4 虚函数

在现实生活中，经常出现这种情况：面对同样的消息，不同的接收者会有不同的反应。例如，国家发布税收新政策“年收入 12 万以上者自行申报纳税”这条消息，税务局的人收到这条消息，会安排窗口等待满足条件的人前来纳税；而普通民众则会计算自己的年收入，看是否满足条件，决定是否前往税务局纳税。不同的人，面对同样的消息，产生不同的反应。

面向对象语言是解决现实世界问题的，也需要对这种实际情况进行处理。C++程序设计语言使用多态性实现同一个消息，不同接收者采取不同的响应方式的这种现象。

顾名思义，多态性是一个事务有多种形态。在面向对象语言中，一般这样描述多态：向不同对象发送同一个消息，不同的对象在接收时会产生不同的行为（即方法）。也就是说，每个对象可以用自己的方式去响应共同的消息。



C++语言的多态性有两种类型：静态多态性和动态多态性。前面学习的函数重载和运算符重载就是静态多态性的具体示例。由于在程序编译时，系统就能够决定调用哪个函数，因此静态多态性又称为编译时的多态性。动态多态性在程序运行过程中才动态地确定操作所针对的对象，它又称为运行时的多态性。动态多态性是通过虚函数实现的。

10.4.1 静态多态性

对于普通成员函数的重载，可表达为下面的方式：

- ① 在同一个类中重载；
- ② 在不同类中重载；
- ③ 基类的成员函数在派生类中重载。

因此，重载函数的访问是在编译时区分的，这种程序运行之前就能够在多个函数中确定当前访问的函数的方法称为静态多态性。有以下三种区分方法。

- ① 根据参数的特征加以区分，例如：

Show(int, char)与 Show(char *, float)

不是同一函数，编译能区分。

- ② 使用“::”加以区分，例如：

Circle :: Show 有别于 Point :: Show

- ③ 根据类对象加以区分，例如：

ACircle.Show() 调用 Circle :: Show()

APoint.Show() 调用 Point :: Show()

这里，ACircle 和 APoint 分别是 Circle 和 Point 的对象。

例 10-20 显示基类的成员函数在派生类中重载时的区分方法。

【例 10-20】

```
#include <iostream.h>
class A
{
public:
    void fun(){cout<<"In A"<<endl;}
};
class B:public A
{
public:
    void fun(){cout<<"In B"<<endl;}
};
class C:public B
{
public:
```

```

        void fun(){cout<<"In C"<<endl;}
};
void main()
{
    C Cobj;
    Cobj.fun();           //调用 C::fun()
    Cobj.B::fun();        //调用 B::fun()
    Cobj.A::fun();        //调用 A::fun()
    A & Aref=Cobj;
    Aref.fun();           //调用 A::fun()
    B & Bref=Cobj;
    Bref.fun();           //调用 B::fun()
    Bref.B::fun();        //调用 B::fun()
    A * Apointer=&Cobj;
    Apointer->fun();       //调用 A::fun()
}

```

可以看到，派生类对象调用的 `Cobj.fun()` 是派生类自己的 `fun()` 函数，基类的 `fun()` 函数被派生类对象的 `fun()` 函数屏蔽了，因此不能直接被派生类对象调用。如果要通过派生类对象访问被屏蔽的基类同名函数，只能通过对象指定基类名字进行访问，例如，访问 `B` 基类的 `fun()` 函数，需指定 `Cobj.B::fun()`；访问 `A` 基类的 `fun()` 函数，指定 `Cobj.A::fun()`。

如果用基类类型给派生类指定别名（`A & Aref=Cobj`），则通过别名访问的 `fun()` 函数（`Aref.fun()`）是基类的函数（调用 `A::fun()`）。如果用基类指针指向派生类对象（即 `A * Apointer=&Cobj`），则通过基类指针访问的 `fun()` 函数（`Apointer->fun()`）是基类的函数（调用 `A::fun()`）。

因此，如果类层次有函数重载，派生类对象访问的是派生类自己的成员函数，但基类引用或者基类指针和派生类对象绑定之后，通过基类引用或者基类指针访问的虽然是派生类对象，但访问的重载函数仍然是基类自己的成员函数。

那么，如果用指向基类的指针指向派生类对象，或者用指向派生类的指针指向基类，有什么样的情况出现？

10.4.2 基类和派生类的指针与对象的关系

通过例 10-21，分析基类和派生类之间的指针与对象的关系。

【例 10-21】

```

#include <iostream.h>
#include <string.h>
class B_class
{

```



```
public :
    void put_name( char * s)
    { strcpy(name, s);}
    void show_name( )
    { cout << name << "\n";}
private:
    char  name [80];
};
class  D_class : public B_class
{
public:
    void put_phone(char * num)
    { strcpy(phone_num,num);}
    void show_phone( )
    { cout << phone_num << "\n";}
private:
    char phone_num [80];
};
main()
{
    B_class * p;
    B_class B_ob;
    D_class * dp;
    D_class D_ob; p = &B_ob;      //基类指针指向基类对象
    p->put_name("Thomas Edison");//访问基类成员函数
    p = &D_ob;                    //基类指针指向派生类对象
    p->put_name("Albert Einstein");//访问基类成员函数
    B_ob.show_name( );           //基类对象访问基类成员函数
    D_ob.show_name( );           //派生类对象访问继承下来的基类成员函数
    dp = &D_ob;                  //派生类指针指向派生类对象
    dp->put_phone("555555_1234");//派生类指向访问派生类成员函数
    dp->show_phone( );
    p->show_phone( );             //基类指针访问派生类成员函数，不存在，错误
    ((D_class*)p)->show_phone( );//基类指针强制转换为派生类指针访问派生类成员函数
    p->show_phone( );             //错误
}
```

该程序输出如下：

Thomas Edison

Albert Einstein

555555_1234

555555_1234

注意:

① 可以用一个指向基类的指针指向其公有派生类的对象。这时,基类指针访问的是派生对象的拥有的基类部分,派生类自身的部分不能被基类指针访问。但是,用指向派生类的指针指向一个基类的对象是不正确的,因为派生类指针可以访问派生类公有成员,但基类对象没有派生类自身成员,因此用派生类指针指向基类对象时,如果访问派生类公有成员,而该成员实际不存在,所以不能用指向派生类的指针指向一个基类的对象。

② 希望用基类指针访问其公有派生类的特定成员,必须将基类指针用显示类型转换为派生类指针。例如:

```
((D_class* )p) -> show_phone( );
```

外层括号表示是对 p 的强制转换,而不是返回类型。

10.4.3 虚函数与多态性

1. 虚函数的概念

一个指向基类的指针可用来指向从基类公有派生的任何对象,这一事实是非常重要的,是 C++语言实现运行时多态性的关键途径。如果有多个或者多层派生类,通过一个基类指针可以访问所有派生类对象的成员函数,这样就可以达到一个接口、多个实现的访问了。分析例 10-22,基类指针能否访问不同派生类对象的成员函数呢?如果不能,采取什么样的方式可以达到这个目的呢?

【例 10-22】

```
#include <iostream.h>

class Base
{
public:
    Base(int a)
    {x=a;}

    void who()
    { cout << "base  "<<x<<"\n";}

protected:
    int x;
};

class First_d: public Base
{
```



```
public:
    First_d (int a ):Base(a)
    { }
    void who(){ cout << "First derivation  " <<x<<"\n";}
};
class Second_d :public Base
{
public:
    Second_d (int a):Base(a)
    { }
    void who(){ cout << "Second derivation  " <<x<<"\n";}
};
void main()
{
    Base * p;
    Base base_obj(1);
    First_d first_obj(2);
    Second_d second_obj(3);
    p=&base_obj;
    p->who();
    p=&first_obj;
    p->who();
    p=&second_obj;
    p->who();
    first_obj.who();
    second_obj.who();
}
```

该程序的输出为：

```
base  1
base  2
base  3
First derivation  2
Second derivation 3
```

指向基类的指针 `p`，不管是指向基类的对象 `base_obj` 还是指向派生类的对象 `first_obj` 和 `second_obj`，`p->who()` 调用的都是基类定义的 `who()` 的版本。这说明，通过指针引起的普通成员函数调用，仅仅与指针（或引用）的类型有关，而与此刻正在指向什么对象无关。因为基类的指针仅能访问派生类中继承的基类成员，而不能访问派生类自己的成员。在这种情况下，必须显式地用 `first_obj.who()` 和 `second_obj.who()` 才能调用类 `first_d` 和类 `second_d` 中定义的

who()的版本。其本质的原因在于，普通成员函数的调用是在编译时静态区分的。

如果随着 *p* 所指向的对象的不同，*p->who()*能调用不同类中 who()的版本，这样就可以用一个界面 *p->who()*访问多个实现版本：Base 中的 who()、First_d 中的 who()以及 Second_d 中的 who()。这在编程时非常有用。实际上，这表达了一种动态的性质，函数调用 *P->who()*依赖于运行时 *p* 所指向的对象。虚函数提供的就是这种解释机制。虚函数是在基类中被冠以 virtual 的成员函数，它提供了一种接口界面。虚函数可以在一个或多个派生类中被重新定义，但要求在派生类中重新定义时，虚函数的函数原型，包括返回类型、函数名、参数个数和参数类型的顺序，必须完全相同。如果在 Base 中将成员函数 who()说明为虚函数，则修改例 10-22 的程序如下。

【例 10-23】

```
#include <iostream.h>

class Base
{
public:
    Base(int a)
    {x=a;}
    virtual void who()
    { cout << "base  " <<x<<"\n";}

protected:
    int x;
};

class First_d : public Base
{
public:
    First_d (int a ):Base(a)
    { }
    void who(){ cout << "First derivation  " <<x<<"\n";}
};

class Second_d: public Base
{
public:
    Second_d (int a):Base(a)
    { }
    void who(){ cout << "Second derivation  " <<x<<"\n";}
};

void main()
{
    Base * p;
```



```
Base base_obj(1);
First_d first_obj(2);
Second_d second_obj(3);

p=&base_obj;
p->who();
p=&first_obj;
p->who();
p=&second_obj;
p->who();
first_obj.who();
second_obj.who();
}
```

该程序的输出为：

```
base 1
First derivation 2
Second derivation 3
First derivation 2
Second derivation 3
```

这里，语句 `p->who()` 出现了三次，由于 `p` 指向的对象不同，每次出现都执行了 `who()` 的不同实现版本。基类的虚函数 `who()` 定义了一种接口，在派生类中为此接口定义了不同的实现版本，由于虚函数的解释机制，实现了“单界面、多实现版本”的思想。这种在运行时将函数界面与函数的不同实现版本进行匹配的过程，称为晚期匹配，也称为运行时的多态性。

2. 运行时的多态性与虚特性

(1) 运行时的多态性

晚期匹配如何发生？所有的工作都由编译器在幕后完成。利用 `virtual` 表达晚期匹配要求，编译器自动安装实现晚期匹配所必需的所有机制。这意味着，如果对 `First_d` 或者 `Second_d` 对象通过基类 `Base` 地址调用 `who()`，将得到恰当的函数。

为了完成这件事，编译器对每个包含虚函数的类创建一个表（称为 `VTABLE`）。在 `VTABLE` 中，编译器放置特定类的虚函数地址。在每个带有虚函数的类中，编译器秘密地放置一个指针，称为 `vpointer`（缩写为 `VPTR`），指向这个对象的 `VTABLE`。通过基类指针进行虚函数调用时（也就是进行多态调用时），编译器静态插入取得这个 `VPTR`，并在 `VTABLE` 表中查找函数地址的代码，这样就能调用正确的函数，使晚期匹配发生。

为每个类设置 `VTABLE`，初始化 `VPTR`，为虚函数调用插入代码，所有这些都是自动发生的，所以不必担心这些。利用虚函数，这个对象的合适的函数就能被调用，哪怕在编译器还不知道这个对象的特定类型的情况下。

虽然系统在进行动态关联时的时间开销很少，因此多态性是高效的。但是当一个类带有虚函数时，编译系统会为该类构造一个虚函数表，存放每个虚函数的入口地址，因此占用一定的空间开销。那么，一个成员函数什么时候需要声明为虚函数呢？主要考虑以下 4 点。

- ◎ 首先考虑成员函数所在的类是否会作为基类；然后看成员函数在类的继承后有无功能被修改，如果希望修改其功能，一般将它声明为虚函数。
- ◎ 如果成员函数在类被继承之后功能不需要修改，或派生类中用不到该函数，则不要把它声明为虚函数。
- ◎ 还应当考虑对成员函数的调用是通过对象名还是基类指针或引用去访问。如果通过基类指针或引用去访问，则声明为虚函数。
- ◎ 如果希望通过基类指针或者引用访问派生类成员函数，但基类功能比较抽象或者不能确定功能，可以将基类定义为抽象类，即只定义函数名字，没有函数体，具体功能由派生类添加。

(2) 虚特性

用虚函数实现运行时多态性的关键是，必须用指向基类的指针（或者引用）访问虚函数。尽管可以像调用其他成员函数那样显式地用对象名来调用一个虚函数，但只有在一个指向基类的指针（或者引用）访问虚函数时，运行时多态性才能实现，才可能出现上例 `main` 函数中相同的界面(`p->who();`)出现 3 次的情况，这是因为 `p` 指向的对象不同，而调用了 `who()` 的 3 个不同实现版本。这时，称为函数具有虚特性。

基类函数 `f` 具有虚特性的条件如下。

- ◎ 在基类中，将该函数说明为虚函数。这样可以在派生类中重新定义此函数，为它赋予新的功能，并能够方便调用。在类外定义虚函数时，不必再加 `virtual` 关键字。
- ◎ 定义基类的公有派生类。
- ◎ 在基类的公有派生类中，原型一致地重载该虚函数。
- ◎ 定义指向基类的指针变量，它指向基类的公有派生类的对象（或定义基类的引用，它引用基类的公有派生类的对象）。

注意：在一个派生类中重定义基类的虚函数是函数重载的另一种特殊形式。但它不同于一般的函数重载。当重载一般的函数时，函数的返回类型和参数表可能是不相同的，仅函数名要求相同。但重载一个虚函数时，要求函数名、返回类型、参量个数、参数类型和顺序是完全相同的。如果不同，会产生什么情况呢？

- ◎ 仅仅返回类型不同，其余相同。

C++语言认为这是错误的，因为仅仅返回类型不同的函数本质上是含糊的（在制定 ANSI C++标准的讨论中，许多人要求放松这一限制）。

- ◎ 函数原型不同，仅函数名相同。

C++语言认为这是一般的函数重载，此时虚特性丢失。

以前的函数重载处理的是同一层次上的同名函数问题，而虚函数处理的是不同派生层次的同名函数问题，前者是横向重载，后者是纵向重载。但与重载不同的是：同一类族的虚函数的首部是相同的，而函数重载时函数的首部是不同的（参数个数或者类型不同）。如果纵向重载不使用 `virtual`，则当基类指针指向不同派生对象时，访问的只有基类成员函数；如果用派生类对象访问，则是同名覆盖，即访问的是派生类对象自己的成员函数，参见例 10-24。



【例 10-24】

```
class base
{
public:
    virtual void vf1();
    virtual void vf2();
    virtual void vf3();
    void f();
};

class derived : public base
{
public:
    void vf1();           // 具有虚特性
    void vf2(int);        // 一般函数重载，参数不同，虚特性丢失
    char vf3();           // 错误：仅返回类型不同
    void f();             // 一般的函数重载，非虚函数的重载
};

void g()
{
    derived d;
    base * bp=&d;         // 基类指针指向公有派生类对象
    bp->vf1();             // 调用 derived::vf1()
    bp->vf2();             // 调用 base::vf2()
    bp->f();               // 调用 base::f()
}
```

派生类 `derived` 中的函数 `vf1()` 与基类 `base` 中的虚函数 `vf1()` 具有完全相同的函数原型，故保持了虚特性。而函数 `vf2(int)` 与基类中的虚函数 `vf2()` 参数不同，即函数原型不同，仅函数名相同，这只是一般函数的重载，其虚特性丢失。函数 `char vf3()` 同基类的虚函数 `void vf3()` 相比较，仅返回类型不同，目前的 C++ 语言实现认为这是错误的。函数 `f()` 仅仅是基类非虚函数 `f()` 的重载。

由于 `vf1()` 保持了虚特性，`vf2()` 丢失了虚特性，因此，在进行函数调用时，结果很不一样。在函数 `g()` 中，语句

```
bp->vf1();
```

调用的是 `derived::vf1`。在派生类 `derived` 中，函数 `vf1` 定义为虚函数，虚函数调用的解释依赖于调用它的对象类型，`bp` 虽然是指向基类的指针，但此刻指向的是派生类对象 `d`，因此，该语句等价于

```
d.vf1();
```

另外一条语句

```
bp -> f();
```

调用的却是 `base::f`。函数 `f()` 在基类和派生类中均已定义，且函数原型相同，但它是一个非虚函数。非虚函数调用的解释仅依赖于表示调用它的对象的指针或引用类型。`bp` 被声明为指向基类的指针，非虚函数的调用仅仅依赖 `bp` 是指向基类的指针，而不在乎 `bp` 此刻是否正在指向派生类的对象，它调用的是 `base::f()`。同样地，语句 `bp->vf2()` 调用的是 `base::vf2()`。读者可以将 `bp->vf2()` 改写为 `bp->vf2(1)`，看看结果如何？

虚函数的这种特性使派生类和虚函数成为许多 C++ 程序设计语言的关键，因为基类可以使用虚函数提供一个界面，这是该类的所有公有派生类都具有的共同界面。但派生类可以定义自己的实现版本，而且虚函数调用的解释依赖于调用它的对象类型，使指向基类对象的指针指向不同派生类的对象，就能访问虚函数的不同实现版本。

为什么多种实现版本的一致界面是重要的？答案又回到面向对象的中心目标，帮助程序员控制更大的复杂性。

例 10-25 有助于了解“单界面、多实现版本”的概念。

【例 10-25】

```
#include <iostream.h>

class figure
{
public:
    void set_dim(double i, double j=0)
    {
        x=i;
        y=j;
    }
    virtual void show_area()
    {
        cout << "No area computation defined";
        cout << "for this class.\n";
    }
protected:
    double x,y;
};

class triangle : public figure
{
public:
    void show_area()
    {
        cout << "Triangle with high ";
```



```
        cout << x << " and base " << y;
        cout << " has an area of ";
        cout << x* 0.5* y << "\n";
    }
};

class square : public figure
{
public:
    void show_area()
    {
        cout << "Square with dimension";
        cout << x << "*" << y;
        cout << " has an area of ";
        cout << x * y << "\n";
    }
};

class circle : public figure
{
public:
    void show_area()
    {
        cout << "Circle with radius ";
        cout << x;
        cout << " has an area of ";
        cout << 3.14 * x * x;
    }
};

void main()
{
    figure * p;
    triangle t;
    square s;
    circle c;
    p=&t;
    p->set_dim(10.0, 5.0);
    p->show_area();
    p=&s;
    p->set_dim(10.0, 5.0);
```

```

        p->show_area();
        p=&c;
        p->set_dim(9.0);
        p->show_area();
    }

```

该程序的输出为：

Triangle with high 10 and base 5 has an area of 25.0

Square with dimension 10* 5 has an area of 50.0

Circle with radius 9 has an area of 254.34

考察该程序会发现，各个派生类共享基类函数 `void set_dim(double I, double j=0)`。而对于计算面积，各个实际图形有不同的算法，`square`、`triangle` 和 `circle` 具有相同的界面：`show_area()`，但它们对应不同的实现方法，所以，函数 `show_area()` 定义为虚函数。

注意：

① 虚函数必须是类的成员函数。不能将虚函数说明为全局（非成员的）函数，也不能说明为静态成员函数。不能将友元说明为虚函数，但虚函数可以是另一个类的友元。

② 析构造函数可以是虚函数，但构造函数不能是虚函数。

③ 一旦一个函数被说明为虚函数，不管经历了多少派生类层，都将保持其虚特性。

C++语言规定，当一个成员函数被声明为虚函数之后，其派生类中的同名函数都自动成为虚函数。因此在派生类重新声明该虚函数时，可以加 `virtual`，也可以不加，但习惯上一般在每层声明该函数时都加上 `virtual`，使程序更加清晰。在虚函数概念中的例子，若 `second_d` 由 `first_d` 派生，而不是由 `Base` 派生，则 `who()` 仍是虚函数，并定义了自己的版本，例如：

```

class Second_d: public First_d
{
    public:
        void who() {cout<<"second derivation\n"<<x<<"\n";
}

```

当一个派生类没有重新定义虚函数时，则使用其基类的虚函数版本。因此，在使用时要记住继承的层次性。

3. 虚析构造函数

分析例 10-26 程序的输出。

【例 10-26】

```

#include <iostream>
using namespace std;
class base
{
    public:
        base(){cout<<"base()"<<endl;}
}

```



```
~base(){cout<<"~base()"<<endl;}

};

class derived:public base
{
public:
    derived(){cout<<"derived()"<<endl;}
    ~derived(){cout<<"~derived()"<<endl;}

};

int main()
{
    base *pb=new derived;
    delete pb;
    return 0;
}
```

程序运行结果为：

```
base()
derived();
~base();
```

构造函数 `base()` 和 `derived()` 都被调用了，但是析构函数只调用了 `~base()`。生成了派生类对象，但释放的时候却没有调用派生类析构函数完成清理工作。因为基类指针只调用基类成员函数，不能够调用派生类成员函数，即使析构函数也是如此。如果希望能够执行派生类的析构函数，则需要将基类的析构函数声明为虚析构函数。即，将

```
~base(){cout<<"~base()"<<endl;}
```

修改为：

```
virtual ~base(){cout<<"~base()"<<endl;}
```

则执行修改后的例 10-26 得到结果如下：

```
base()
derived();
~derived();
~base();
```

先调用派生类的析构函数，再调用基类的析构函数，满足人们的期望。当基类的析构函数声明为虚函数时，无论指针指向的是同一类族中的哪一个对象，当对象撤销时，系统会采用动态关联，调用相应的析构函数，对该对象进行清理工作。

在程序中，最好将所有的析构函数都声明为虚析构函数，这将使所有的派生类的析构函数也都自动成为虚函数。这样就不用担心出现可能不能调用派生类的析构函数的情况。即使基类不需要析构函数进行清理工作，但也常常显式地定义一个函数体为空的虚析构函数，以保证在撤销动态存储空间时得到正确的处理。

构造函数不能声明为虚函数，因为在执行构造函数时，类对象还没有完全建立，不能使

用函数与类对象的动态关联。



10.5 纯虚函数和抽象类

基类往往表示一些抽象的概念。例如，`shape` 是一个基类，它表示具有形状的东西，从 `shape` 可以派生出封闭图形和非封闭图形两个派生类。封闭图形又可以派生出椭圆形、多边形……这个类等级的基类 `shape` 体现了一个抽象的概念，在 `shape` 中定义一个求面积的函数显然是无意义的，但可以将其说明为虚函数，提供各派生类一个公共的界面，并由各派生类提供求面积函数的各自版本。在这类情况下，基类的有些虚函数没有定义是很正常的，但是要求派生类必须重定义这些虚函数，以使派生类有意义。为此，C++语言引入了纯虚函数的概念。

纯虚函数是一个在基类中说明的虚函数，它在该基类中没有定义，要求任何派生类都必须定义自己的版本。为说明一纯虚函数，使用下列一般形式：

```
virtual type func_name(参数表)=0;
```

其中，`type` 是函数的返回类型，`func_name` 是函数名。

将一虚函数声明为纯虚函数，就要求任何派生类都应该定义自己的实现。在构造函数和析构函数中调用虚函数时，采用静态联编，因此，在构造函数和析构函数中不能够调用纯虚函数。但其他的成员函数可以调用纯虚函数。

如果一个类至少有一个纯虚函数，那么就称该类为抽象类。抽象类机制支持一般概念的表达。例如上面谈到的形状类 `shape` 是一般的概念，可以表达为抽象类，它有许多具体的变种，如圆形和方形才是具体可使用的类。抽象类也可用于定义接口，由派生类提供各种实现。

抽象类只能用做其他类的基类，抽象类不能建立对象。抽象类不能用做参数类型、函数返回类型或显式转换的类型，但可以声明抽象类的指针和引用，参见例 10-27。

【例 10-27】

```
class point { ... };
class shape
{
    point center;
    ...
public:
    point where( )
    { return center; }
    void move( point p )
    { center=p;draw( );}
    virtual void rotate( int )=0;
    virtual void draw( )=0;
};
```



那么：

```
shape x;           // 错误：抽象类不能建立对象
shape * p;         // 可以声明抽象类的指针
shape f ();        // 错误：抽象类不能作为返回类型
void g(shape);     // 错误：抽象类不能作为参数类型
shape & h(shape &); // 可以声明抽象类的引用
```

从基类继承来的纯虚函数，在派生类中仍是纯虚函数，除非派生类按原型一致地重载该纯虚函数。

例如：

```
class ab_circle : public shape
{
    int radius;
public:
    void rotate(int) { ... }
};
```

由于 `shape::draw()` 是一个纯虚函数，默认的 `ab_circle::draw()` 也是一个纯虚函数，这时 `ab_circle` 仍为抽象类。要使 `ab_circle` 类为非抽象的，必须进行如下说明：

```
class ab_circle : public shape
{
    int radius;
public:
    void rotate(int) { ... }
    void draw() { ... }
};
```



10.6 虚函数实例——Figure类

本节举一个更加完整的例子，建立一个模块，该模块表示任意形状的图形类，具备在屏幕上移动任意图形的功能，类名为 **Figure**。设计 **Figure** 模块的主要目标是，允许不同图形的用户来扩充模块中定义的类，并且还可以不同形状的图形来使用模块的所有特征。建立在屏幕上移动任意图形的方法，以响应用户的输入。

作为一个最初的方法，可能考虑 **Drag** 函数，它把一个对象用做一个参数，然后围绕屏幕移动该对象，参见例 10-28。

【例 10-28】

```
void Drag( Point & Anyfigure , int DragBy)
{
    int Deltax, Deltay;
    int Figurex, Figurey;
```



```

Anyfigure.Show( );
Figurex=Anyfigure.Getx( );
Figurey=Anyfigure.Gety( );
while( GetDelta( Deltax, Deltay ))
{
    Figurex=Figurex+(Deltax* DragBy);
    Figurey=Figurey+(Deltay* DragBy);
    Anyfigure.MoveTo( Figurex, Figurey);
}
}

```

注意:

Anyfigure 被说明为具有 Point&类型, 它的含义是“对 Point 类型的一个对象的引用”。这说明, Point 类型的对象或任何 Point 派生类型的对象都可作为 Anyfigure 的引用参数传递。因此, Point 或 Circle 类型的对象, 或者将来要定义的从 Point 或 Circle 派生来的类型都可以毫无困难地传递到 Anyfigure 中。

Drag 调用一个这里没有出现的辅助函数 GetDelta, 它从用户那里接收各种 x 和 y 的变化量, 这可以从键盘、鼠标或操纵杆接收到(为简单起见, 从键盘接收输入)。

变量 DragBy 中存放的是每个移动单位(这里是每个键)代表多少个像素点。这个 Drag 函数是非常一般化的, 对于从 Point 类派生出来的任何图形对象, 它都能在 GetDelta(Deltax, Deltay)函数接收的输入控制下, 在屏幕上移动该对象。移动该对象的动作是由语句 Anyfigure.MoveTo(Figurex, Figurey); 控制的。它的含义是: 图形对象 Anyfigure 根据自己类提供的成员函数 MoveTo, 将图形移到新的位置(Figurex, Figurey)上。这个 Drag 函数能很好地工作。现在的问题是, 能否将 Drag 作为一个成员函数, 放在类等级的某个地方, 由派生类对象共享。

怎样向一个已存在的类等级中增加一个新的成员函数, 这个成员函数应该放在类等级中的什么地方? 首先考虑由这个函数提供的功能, 然后决定这种功能的应用广度。移动一个图形包括从用户那儿得到响应后变动这个图形的位置。从继承的角度, 它正好处于 MoveTo 的地位——适用于 MoveTo 的任何对象也能继承 Drag。因此, Drag 应该成为与 MoveTo 同级的成员函数, 以便 Point 派生的所有类型可以共享它。

在类等级中解决了 Drag 的位置以后, 就可以仔细研究一下它的定义了。作为基类 Point 的一个成员函数, 无须对 Point& Anyfigure 进行显式引用, Point 类的派生类的对象可以以继承方式使用 Drag 函数。因此, Drag 函数原型变成:

```
void Drag(int DragBy)
```

由于去掉了 Anyfigure 参数, 因此, 必须重写 Drag, 以便它所调用的函数, 如 Getx、Gety、Show、MoveTo 和 Hide, 能正确地引用所要移动对象的类型的版本。与前面见到的一样: 函数 Show 和 Hide 需要特殊的和形体有关的代码, 因此可说明成虚函数。MoveTo 函数主要调用 Show()和 Hide(), 通过继承关系, 它能正确调用 Show()和 Hide()的版本。Getx



和 `Gety` 没什么问题，它们只是简单地返回位置信息。

剩下的决策是 `Drag` 应说明为虚函数吗？使一个函数成为虚函数的条件是，这个函数的实现版本在类等级中是否需要动态确定。对目前一些简单的图形来说，`Drag` 够用了，但也许将来有这样的类，例如，在 `CAD` 应用中可能需要一个特别的移动动作，也许移动一个等距图形将需要一些比例操作等。因此，将 `Drag` 说明为虚函数。具体参见例 10-29。

【例 10-29】

```
//figures.h
enum Boolean {false, true};
class Location
{
public:
    Location(int InitX, int InitY)    {X = InitX;Y = InitY;}
    int GetX() {return X;}
    int GetY() {return Y;}
protected:
    int X;  int Y;
};
class Point : public Location
{
public:
    Point(int InitX, int InitY);
    virtual void Show();
    virtual void Hide();
    virtual void Drag(int DragBy);
    Boolean IsVisible() {return Visible;}
    void MoveTo(int NewX, int NewY);
protected:
    Boolean Visible;
};
class Circle : public Point
{
protected:
    int Radius;
public:
    Circle(int InitX, int InitY, int InitRadius);
    void Show();
    void Hide();
    void Expand(int ExpandBy);
```

```

        void Contract(int ContractBy);
    };
    Boolean GetDelta(int& DeltaX, int& DeltaY);
// FIGURES.CPP
#include "figures.h"
#include <graphics.h>
#include <conio.h>
Point::Point(int InitX, int InitY) : Location(InitX, InitY)
{
    Visible = false;
}
void Point::Show()
{
    Visible = true;
    putpixel(X, Y, getcolor());
}
void Point::Hide()
{
    Visible = false;
    putpixel(X, Y, getbkcolor());
}
void Point::MoveTo(int NewX, int NewY)
{
    Hide();
    X = NewX;
    Y = NewY;
    Show();
}
Boolean GetDelta(int& DeltaX, int& DeltaY)
{
    char KeyChar;
    Boolean Quit;
    DeltaX = 0;
    DeltaY = 0;
do
{
    KeyChar = getch();
    if(KeyChar == 13)

```



```
        return(false);
    if(KeyChar == 0)
    {
        Quit = true;
        KeyChar = getch();
        switch(KeyChar)
        {
            case 72: DeltaY = -1;break;
            case 80: DeltaY = 1;break;
            case 75: DeltaX = -1;break;
            case 77: DeltaX = 1;break;
            default: Quit = false;
        }
    }
    while(!Quit);
    return(true);
}

void Point::Drag(int DragBy)
{
    int DeltaX, DeltaY;
    int FigureX, FigureY;
    Show();
    FigureX = GetX();
    FigureY = GetY();
    while(GetDelta(DeltaX, DeltaY))
    {
        FigureX +=(DeltaX * DragBy);
        FigureY +=(DeltaY * DragBy);
        MoveTo(FigureX, FigureY);
    }
}

Circle::Circle(int InitX, int InitY, int InitRadius) : Point(InitX, InitY)
{
    Radius = InitRadius;
}

void Circle::Show()
{

```

```

        Visible = true;
        circle(X, Y, Radius);
    }
void Circle::Hide()
{
    unsigned int TempColor;
    TempColor = getcolor();
    setcolor(getbkcolor());
    Visible = false;
    circle(X, Y, Radius);
    setcolor(TempColor);
}
void Circle::Expand(int ExpandBy)
{
    Hide();
    Radius += ExpandBy;
    if(Radius < 0)
        Radius = 0;    Show();
}
void Circle::Contract(int ContractBy)
{
    Expand(-ContractBy);
}
// FIGDEMO.CPP
#include "figures.h"
#include <graphics.h>
#include <conio.h>
class Arc : public Circle
{
    int StartAngle;
    int EndAngle;
public :
    Arc(int InitX, int InitY, int InitRadius, int InitStartAngle, int InitEndAngle) : Circle(InitX,
        InitY, InitRadius)
    {
        StartAngle = InitStartAngle;
        EndAngle = InitEndAngle;
    }

```



```
    }  
    void Show( );  
    void Hide( );  
};  
  
void Arc :: Show( )  
{  
    visible = true;  
    Arc ( X, Y, StartAngle, EndAngle, Radius );  
}  
void Arc :: Hide( )  
{  
    int TempColor;  
    TempColor = getcolor( );  
    visible = false;  
    Arc(X,Y,StartAngle, EndAngle, Radius );  
    setcolor( TempColor);  
}
```

整个模块的类等级为：

```
Location  
↑  
Point  
↑  
Circle  
↑  
Arc
```

其中，Circle 和 Arc 是两个实用类。

下面的函数说明了这个类等级的使用。它的功能是，先画一个圆，再画一条弧，然后移动这条弧。当接收到回车键时，移动结束。抹去这条弧，然后移动这个圆，按回车键，移动结束。

```
void main( )  
{  
    int graphdriver = DETECT, graphmode;  
    initgraph(& graphdriver, & graphmode, " ...\\ bgi");  
    Circle ACircle(151, 82, 50);  
    Arc AnArc(151, 82, 25, 0, 190);  
    AnArc.Drag(5);  
    AnArc.Hide();  
}
```

```

    ACircle.Drag(10);
    closegraph();
}

```



10.7 类属

类属（Genericity）首先由 ALGOL 68 引入，是 Ada 语言的典型成分。关于 Ada 的类属，只考虑它的一种最重要的形式，即类型参数化，它表现了使用一个或多个类型去参数化一个软件元素（如 Ada 中的程序包或函数）的能力。类属又可分为无约束类属机制和约束类属机制。其中，无约束类属机制是指对类属参数没有施加任何特殊的要求，而约束类属机制则意味着类属参数需要一定的辅助条件。

10.7.1 无约束类属机制

考虑一个函数，它用来交换两个变量的值。使用非静态强类型语言，可编写出如下函数（用类 Ada 语言的语法形式）：

```

procedure swap(x, y) is
  t : local;
begin
  t := x; x := y; y := t;
end swap;

```

这里，被交换的元素 x 和 y 的类型以及局部变量 t 的类型都不需要指定，这显得很自由，但可能会导致错误。如果 a 是整型变量， b 是字符串，则 $\text{swap}(a,b)$ 显然会引起错误，而编译程序却无法检查。

为了解决这一问题，像 Pascal 这样的静态强类型语言需要程序员明确地定义所有变量和形参的类型，迫使函数调用时进行实参与形参之间的强类型检查，以避免产生类型不兼容的错误。这就产生了不愉快的后果，在缺少重载机制支持的情况下，要为每种类型的交换操作声明一个新的过程： int_swap ， str_swap ， float_swap ，…。

相比之下，含类属机制的这类语言提供了一种折中的办法，它既不像非类型语言那样有太多的自由，也不像 Pascal 那样的强类型语言施加太多的约束。一个类属化的 swap 可以用类 Ada 语言声明为：

```

generic
  type T is private;
procedure swap(x, y: in out T) is
  t : T;
begin
  t:=x; x:=y; y:=t;
end swap

```



语句 `generic` 引入了一个类型参数 `T`，也称类属参数 `T`，`swap` 的两个形参 `x`、`y` 和局部变量 `t` 都具有 `T` 的类型，这样，只要类属参数 `T` 实例化为某一具体类型，例如，`integer` 类型或者 `string` 类型，`swap` 函数就能正确工作。

可见，这里声明的 `swap` 函数并不是一个实际的函数，它只表示一个函数模板，或称类属函数，它代表的是一类函数，实际的函数是将 `T` 实例化而获得的。例如：

```
procedure int_swap is new swap (integer);
```

```
procedure str_swap is new swap (string);
```

这里的类型 `integer` 和类型 `string` 称为类属实参。`int_swap` 对整数类型进行交换操作，`str_swap` 对串类型进行交换操作。这里只需对函数模板进行实例化，无须重新定义新的函数体。

这种类属机制可以称为无约束类属机制。所谓无约束，是指对于作为类属的参数，没有施加任何特殊的条件。这里，可以交换任意类型的变量。

还有另外一种形式的类属定义，它们只有在类属实参满足一定的条件时才有意义，这种形式的类属机制称为约束类属机制。

10.7.2 约束类属机制

考虑下面的例子，假定需要一个类属函数来计算两个值的最小值。其类属程序如下：

```
generic
    type T is private;
function minimum (x,y:T) return T is
begin
    if x <= y then return x;
    else return y endif;
end minimum
```

在这个函数声明中，比较运算符“`<=`”因类型 `T` 不同，它的意义也不相同。显然，结构变量的比较与整型变量的比较有很大的差别。类型 `T` 中必须定义比较运算符“`<=`”，这个函数声明才有意义。关于这一特性的检查只有在允许时才进行。因此，需要按某种方式来指明在类型 `T` 中必须具备有关的操作。

在 `Ada` 语言中，采用如下方式解决这一问题，即把运算符“`<=`”作为类属程序单元自身的类属参数来对待。

```
generic
    type T is private;
    with function "<=" (a,b:T) return
boolean is <>;
function minimum (x,y:T) return T is
begin
    if x <= y then return x;
```



```

else return y endif;

end minimum

```

其中，关键词 **with** 引入函数的类属参数，这里是 “<=”。因此，对任意类型 **T**，可以定义 **minimum** 的一个实例，但还需要提供类属参数 “<=” 的实例（这里视为 “<=” 的实例函数）。假定 “<=” 的实例函数为 **T_le**（如 **integer_le**），这样，实际函数的声明为：

```
function T_minimum is new minimum (integer,integer_le)
```

可见，类属函数 **minimum** 实例化为一实际函数时，需要两个类属实参：**integer** 和 **integer_le**，其中 **integer_le** 是 **integer** 的约束条件。这种形式的类属机制称为约束类属机制。作为类属实参的类型 **integer**，必须具备 **integer_le** 的条件。

以上所述的类属机制称为显式类属机制，其参数化过程是毫不隐讳的。还有一种隐式类属机制，如 **ML** 语言的类属机制，限于篇幅，这里不再介绍。



10.8 模板的概念

近期实现的 C++ 语言也具有类属机制，它被称为模板。模板的功能比较强，借助于 C++ 语言中运算符重载等功能，模板实现了无约束类属机制和约束类属机制。C++ 语言模板允许用户构造类属函数和类属类，也称为模板函数和模板类。

10.8.1 函数模板与模板函数

1. 返回两个参数中较大者

这里，考虑返回两个参数中较大者的函数 **max(x,y)**。**x** 和 **y** 为具有可比较次序的任何类型。但是，因为 C++ 是强类型语言，所以它希望参数 **x** 和 **y** 的类型在编译时就声明。如果不用模板，就需要 **max()** 的许多重载版本，在这些重载版本中，每个版本的代码是相同的，但是形参代表的数据类型却不相同。例如：

```

int max(int x, int y)
{
    return (x > y) ? x: y;
}

long max(long x,long y)
{
    return (x > y) ? x: y;
}

...

```

解决这个问题的一种方法是使用宏：

```
#define max(x, y) (((x)>(y)) ? (x) : (y))
```

然而，使用 **#define** 避开了类型检查机制，使得宏替换在用户不希望其发生的地方出现，例如，使用宏可能会导致在一个 **int** 参数和一个 **struct** 参数之间进行比较。

另一种解决办法是使用模板。如果使用模板，数据类型本身就是一个参数，例如，**max**



函数的模板可以定义为：

```
template <class T>
T max(T x,T y)
{
    return (x>y)? x :y;
}
```

关键字 **template** 表示正在声明一个模板，数据类型参数 **T** 由模板参数 **<class T>** 给出。该模板的含义为，无论模板参数 **T** 实例为 **int**、**char** 或任意其他类型，包括类类型时，函数 **max** 就为实例化了的类型的参数求最大值。这样定义的 **max** 还是函数，但它代表的是一类函数，如果要这个 **max** 函数真正进行求最大值的操作，首先必须将模板参数 **T** 实例化，从这个意义上说，这里定义的 **max** 函数并不是一个完全的函数。称之为函数模板。因此，函数模板代表了一类函数，而且它不是一个完全的函数，必须将其模板参数 **T** 实例化后，才能完成具体函数的功能。将 **T** 实例化的参数常常称为模板实参。用模板实参实例化的函数称为模板函数，例如：

```
void func()
{
    int num1;
    Myclass obj1,obj2;
    int num2=max(num1,0);    // 模板实参为整数类型
    Myclass obj=max(obj1,obj2); // 模板实参为 Myclass 类型
}
```

这里生成了两个模板函数 **max(num1,0)** 和 **max(obj1,obj2)**。**max(num1,0)** 用模板实参 **int** 将类型参数 **T** 实例化，而 **max(obj1,obj2)** 将 **T** 实例化为类类型 **Myclass**。编译器将根据上下文关系非常仔细地调用适当的 **operator>()** 函数，根据运算符重载的知识，在 **Myclass** 类声明中，应提供一个重载的 **operator>()** 函数，以便能正确地执行“>”运算。或者在 **Myclass** 类中提供类类型转换函数 **operator int()**，可以将 **Myclass** 类的对象转换为整数，也能够适应“>”运算。

一个函数模板提供一类函数的抽象，它以任意类型 **T** 为参数。由一个函数模板产生的函数称为模板函数，它是函数模板的具体实例。就像类与对象一样，函数模板将具有相同程序正文的一类函数抽象出来，可以适应任意类型 **T**。函数模板对某一特定类型的实例就是模板函数。函数模板代表了一类函数，模板函数表示某一具体的函数。图 10-13 给出了函数模板和模板函数的关系。

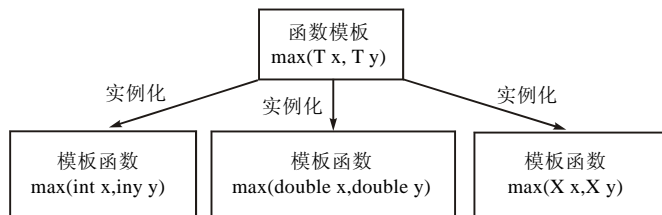


图 10-13 函数模板与模板函数的对应关系

2. 重载模板函数

有些特殊情况需要函数模板参与重载，C++语言允许函数模板被一个或多个同名的非模板函数重载。考虑下面的例子：

```
template <class T>
T max(T a, T b)
{
    return (a>b)? a : b;
}

void fun(int num, char cha)
{
    max(num, num );
    max(cha, cha);
    max(num, cha);    // 错误, max(int, char) 无法匹配
    max(cha, num);    // 错误, max(int, char) 无法匹配
}
```

这里出现了错误。问题在于，模板类型并不知道 `int` 和 `char` 之间能进行隐式类型转换。但是，这样的转换在 C++语言中是很普遍的。

为了解决这个问题，C++语言允许一个函数模板可以使用多个模板参数或者超载一个函数模板，例如：

```
template <class T, class D>
T max(T a, D b)
{
    return (a>b)? a : b;
}
```

用户可以用非模板函数重载一个同名的函数模板，可以这样定义：

```
template <class T>
T max(T a,T b)
{
    return (a>b) ? a : b;
}

int max(int,int);           // 显式地声明 max(int,int)
                             // 这是一个重载的非模板函数

void f(int num,char cha)
{
    max(num,num);           // 调用 max(T, T)
    max(cha,cha);           // 调用 max(T, T)
    max(num,cha);           // 调用 max(int, int)
    max(cha,num);           //调用 max(int, int)
```



```
}
```

这里，非模板函数 `max(int,int)`重载了上述的函数模板 `max(T a,T b)`，当出现调用语句 `max(num,cha);`和 `max(cha,num);`时，它执行的是重载的非模板函数的版本 `max(int,int)`。

还可以定义如下函数：

```
char * max(char * x,char * y)
{
    return (strcmp(x,y)>0) ? x : y;
}
```

非模板函数 `max(char *,char *)`也重载了上述的函数模板。当出现调用语句 `max(a, b);`时，它执行的是这个重载的非模板函数的版本，其中 `a` 和 `b` 都是字符串变量。

在 C++语言中，函数模板与同名的非模板函数的重载方法遵循下述约定：

- ① 寻找一个参数完全匹配的函数，如果找到了，就调用它；
- ② 寻找一个函数模板，将其实例化产生一个匹配的模板函数，如果找到了，就调用它；
- ③ 试一试低一级的对函数的重载方法，例如，通过类型转换可产生参数匹配等，如果找到了，就调用它。

如果①、②、③均未找到匹配的函数，那么这个调用就是一个错误。

如果在第①步有多于一个的选择，那么这个调用是意义不明确的，并且是一个错误。

以上重载模板函数的规则可能会引起许多不必要的函数定义的产生，但一个好的实现，应该是充分利用这个功能的简单性来抑制不合逻辑的回答。

10.8.2 类模板与模板类

一个类模板（也称类属类或类生成类）允许用户为类定义一种模式，参见例 10-30。

【例 10-30】

```
class INTEGER
{
    int item;
public:
    INTEGER(int anitem)
    { item=anitem;}
    void set_item(int anitem)
    { item=anitem;}
    int get_item()
    { return item;}
    ...
};

void main()
{
```

```

        INTEGER Obj(20);
        Obj.set_item(120);
        Obj.get_item( );
        ...
    }

```

类 `INTEGER` 是一个整数类，如果还需要定义一个浮点数类，那么，有：

```

class REAL
{
    float item;
public:
    REAL(float anitem)
    { item=anitem;}
    void set_item(float anitem)
    { item=anitem;}
    float get_item( )
    { return item;}
    ...
};

void main( )
{
    REAL Obj(2.0);
    Obj.set_item(12.0);
    Obj.get_item( );
    ...
}

```

如果还需要定义字符数据类等，那就还要定义许多的类，但它们都很类似，仅仅是处理的数据的类型不同而已。可以定义类模板来解决这个重复现象，参见例 10-31。

【例 10-31】

```

template <class T>
class TemClass
{
    T item;
public:
    TemClass(T anitem)
    { item=anitem;}
    void set_item(T anitem)
    { item=anitem;}
    T get_item( )

```



```
{ return item;}
...
};
void main( )
{
    TemClass<int>  Objint(20);
    TemClass<float>  Objfloat(2.5);
    TemClass<char>  Objchar('A');
    Objint.set_item(120);
    Objchar.get_item( );
    ...
}
```

TemClass 是类模板，完整的名字为 TemClass<T>，在定义对象时，需要实例化类型参数 T，从而生成实际的类，称之为模板类。模板类的名字为 TemClass<TYPE>，如 TemClass<float>等。典型的类模板参见例 10-32。

【例 10-32】考虑一个向量类（一维数组）的例子。不管整型向量还是任何其他类型的向量，在所有类型上进行的基本操作都是相同的（如插入、删除、检索等）。由于将每个元素的类型当做一个类的类型参数来对待，系统可快速生成类型安全的类定义。

类模板定义如下：

```
#include <iostream.h>
template <class T>
class vector
{
    T * data;
    int size;
public:
    vector(int);
    ~vector()
    { delete [] data; }
    T & operator [] (int num)
    { return date[num];}
};
```

其中，关键字 template 表示正在声明一个模板，其类属参数 T 由模板参数<class T>给出。这时，类 vector 声明了一个数组类，当 T 被实例化为 int、char、float、string 或 complex，甚至任意类类型 Myclass 时，vector 被实例化为整型数组、字符数组、浮点数数组、串数组或复数的数组，甚至任意类类型 Myclass 的数组。从这个意义上说，vector 是一个不完全的类。这时，vector 称为类模板，vector<T>是该类模板的名字。将模板参数实例化的参数常称为模板实参，如 int、char 等。用模板实参生成的类称为模板类。例如，vector<int>是一个模板类。

因此，类外成员函数定义的语法为：

```
template <class T>
vector<T>:: vector (int num)
{
    data=new T[num];
    size=num;
}
```

每个模板外的成员函数定义都由 `template <class T>` 开始，表示这是一个类模板的成员函数，其类模板的类名为 `vector<T>`。

下面的 `main` 程序说明了如何使用类模板。

```
void main( )
{
    vector<int> vec(5);        // 产生一个整型向量
    for (int num=0; num<5;++num)
        vec[num]=num;
    for ( num=0; num<5; ++num)
        cout << vec[num] << " ";
    cout << endl;
}
```

输出为：

```
0 1 2 3 4
```

语句 `vector<int> vec(5)` 由模板类 `vector<int>` 声明一个对象 `vec`，它是具有 5 个元素的整型向量。由于在 `vector<int>` 中重载了运算符“[]”，所以与一般数组的使用方法没有什么区别。

`vector` 还可以这样用：

```
vector<int>   v1(20);
vector<complex> v2(30);
typedef vector<complex> cvec;
cvec v3(40);
v1[3] = 7;
v2[3] = complex(7, 8);
```

这里，`vector<int>` 和 `vector<complex>` 都是模板类。它说明了一个整型向量 `v1[20]`，两个复数类型的向量 `v2[30]` 和 `v3[40]`。图 10-14 给出了类模板与模板类的关系。

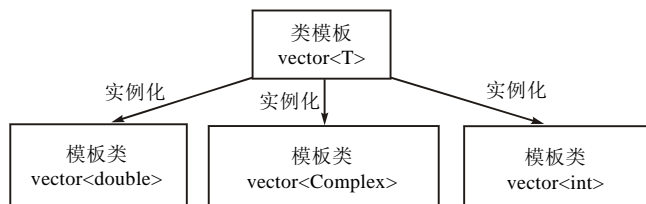


图 10-14 类模板与模板类的对应关系



尽管上述例子都只使用了一个模板参数，但是，实际上允许使用多个参数。模板参数可以是类型，也可以是非类型的数据，例如：

```
template <class T, int size=64>
```

```
class buffer { ... };
```

size 是一个非类型的模板参数，它可以有默认参数。C++语言编译规定，像 size 这样的非类型的模板参数的实参必须是一个常量表达式，例如：

```
const int N=128;
```

```
int num=256;
```

```
Buffer<int, 2*N> b1;
```

```
Buffer<int, num> b2; // 错误, num 不是常量
```

最后一个对象声明是错误的，因为是一个变量，不是常量。这部分规定目前与编译的实现有关，读者使用时一定要查阅相关的手册。

综上所述，一个类模板说明单个类怎样建立，就好像类型声明说明单个对象是怎样建立一样。当用户需要编写几乎相等的代码时，可使用模板。

定义一个模板是一件简单的事。不管是函数模板还是类模板，都必须明确规定其模板参数。因此，模板说明语句：

```
template <class T>
```

中的关键字 **template** 表示正在声明一个模板，模板参数 `<class T>` 规定了类属参数。模板参数可以由一个或多个由逗号隔开的参数组成，参数可以是类型，如 `class T`（`class` 后跟一个标识符组成），也可以是非类型参数（即一般的参数）。

从语法上讲，要说明一个函数模板，只需将模板说明语句放在一个函数说明的前面，并将相应的参数改为模板参数即可。

要说明一个类模板，除了将模板说明语句放在类说明之前以及设置类中相应的模板参数之外，这时类模板的名字总是与模板参数（用尖括号括起来的参数串）一起使用的，例如，`vector<T>` 是类模板名，`vector<int>` 是模板类名。

这里仅仅以直观的方法给出模板的概念和简单的语法，具体使用细节请进一步参阅其他参考文献。

类属和继承的目的都在于提高软件模块的灵活性，取得代码共享的目的。类属是一种更为静态的技术，它采用类型参数化的方法，允许用户为类定义一种模式。类模板的实例化是编译时能确定的，因此，它具有执行速度快可以容纳任意数据类型对象的优点。继承是一种动态技术，它通过扩充和详细说明来支持增量式的模块构造策略，虚函数提供了单界面，多实现版本的多态性。继承机制执行速度慢一些，但提供了更为灵活的动态链接技术。

注意：

即使在定义非内联函数时，模板的头文件中也会放置所有的声明和定义。如果将模板的声明和定义分开放置在不同的文件中，如头文件和实现文件中，在 VC 6.0 中会提示链接出错，如关于堆栈的模板在链接时提示：error LNK2001: unresolved external symbol "public: int __thiscall stack<int>::pop(void)"。但是，把声明和实现放入同一个文件中，链接正确。

这似乎违背了通常的头文件规则：“不要在分配存储空间前放置任何东西”。这条规则是为了防止在连接时的多重定义错误。但模板定义很特殊，由 `template<...>` 处理的任何东西都意味着编译器在当时不为它分配存储空间，它一直处于等待状态，直到被一个模板实例告知。在编译器和连接器的某一处，有一种机制能去掉指定模板的多重定义。所以为了容易使用，几乎总是在头文件中放置全部的模板声明和定义。

有时，也可能为了满足特殊的需要（例如，强制模板实例仅存在于一个简单的 Windows DLL 文件中）而要在一个独立的 CPP 文件中放置模板的定义。大多数编译器有一些机制允许这么做，因此必须检查特定的编译器说明文档以便使用它。



10.9 实例——一维数组

用模板进行程序设计对许多人是陌生的，下面的例子将从一个简单问题入手，介绍怎样将系统的各个部分按概念特性的不同而划分成块（pieces）。当我们研究这些块时，应考虑如何将这里所用的技术也应用到更大的程序中。

这里是一个简单的函数，它将一个整型数组的元素加在一起：

```
int sum(int * p, int n)
{
    int result=0;
    for (int num=0; num<n; num++)
        result+=p[num];
    return result;
}
```

可以这样使用函数：

```
#include <iostream.h>
void main( )
{
    int numb[10];
    for (int num=0; num<10; num++)
        numb[num] = num;
    cout<<sum(numb,10)<<"\n";
    return ;
}
```

这个例子输出 45，它是小于 10 的非负整数的和。

Sum()函数“知道”3 件事：

- ◎ 将一组东西加在一起；
- ◎ 正在相加的东西是整数；
- ◎ 正在相加的整数按一种特殊的方法存储。

我们看一看怎样完善该程序，以使上述每个特性都能用类属的方法来表达。



1. 分离出迭代操作 (iteration)

首先假定要被相加的元素存于数组中，为了遍历数组，即在这个数组上进行迭代操作，我们遵循标准的 C++ 语言技术：使迭代成为一个类。

显然，迭代所需的操作有：

- ◎ 一个构造函数，它建立要被处理的数据；
- ◎ 请求数组中下一个元素的方法；
- ◎ 区分迭代什么时候结束的方法；
- ◎ 赋值运算符函数、复制构造函数和析构函数。

迭代对象常常用于 C++ 类库，它们常常被称为迭代算子 (Iterator)。有许多种迭代算子，这里仅仅给出它们最基本的轮廓。

我们将至今所知道的关于迭代算子的操作写出来：

```
class Int_iterator
{
public:
    Int_iterator(int *, int);
    int valid() const;
    int next();
    Int_iterator(const Int_iterator&);
    Int_iterator& operator=(const Int_iterator&);
    Int_iterator();
};
```

这个类称为 `Int_iterator`，而不是 `Iterator`，因为它确实表达了一个整数序列的迭代，后面将其推广到任意类型。

构造函数有两个参数变元：数组的第一个元素的地址和元素的数目。`Valid()` 成员函数的功能是，如果数组中还有元素，则返回非 0 值；`next` 成员函数在数组中还有元素时取下一个元素，这里假定当数组中没有元素时将不调用 `next`。

在进一步细化迭代算子之前，重写 `sum` 函数如下：

```
int sum(Int_iterator ir)
{
    int result=0;
    while (ir.valid())
        result += ir.next();
    return result;
}
```

使用这个 `sum` 函数的 `main` 程序可能会是这样的：

```
#include <iostream.h>
void main()
{
```

```

    int numb[10];
    for (int num=0; num<10; num++)
        numb[num]=num;
    cout<<sum(Int_iterator(numb,10))<<"\n";
    return;
}

```

唯一的差别是，不是用 `sum(numb,10)`，而是用 `sum(Int_iterator(numb,10))`。当然，也可以写一个重载的函数，它能维持原来的接口界面：

```

inline int sum(int * p, int n)
{
    return sum(Int_iterator(p,n));
}

```

现在回到 `Int_iterator` 类的定义上来，进行进一步细化。这个类的对象实际需要什么信息呢？需要能确定数组中下一个元素的位置，假设就用指针 `int *`，另外还需要确定数组是否结束，这可以用序列中元素的数目来描述。因此，`Int_iterator` 类的私有段成员可描述为：

```

private:
    int * data;
    int len;

```

构造函数直接对私有段两个成员进行初始化：

```
Int_iterator(int * p, int leng):data(p),len(leng) { }
```

析构函数、复制构造函数和类赋值函数就使用系统默认的。

现在 `Int_iterator` 类的描述为：

```

class Int_iterator
{
public:
    Int_iterator(int * p, int leng):data(p), len(leng)
    { }
    int valid( ) const
    {
        return len>0;
    }
    int next( )
    {
        --len;
        return *data++;
    }
private:
    int * data;

```



```
        int len;  
    };
```

2. 在任意类型上进行迭代

名字 `Int_iterator` 非常受限制，它只能描述整数数组的迭代，使用它已失去了抽象的机会。将上述的 `Int_iterator` 类抽象为一般的 `Iterator` 类模板。

```
template <class T>  
class Iterator  
{  
public:  
    Iterator(T * p, int leng):data(p),len(leng)  
    { }  
    int valid( ) const  
    {  
        return len>0;  
    }  
    T next( )  
    {  
        --len;  
        return *data++;  
    }  
private:  
    T * data;  
    int len;  
};
```

这样 `Int_iterator` 模板类可说明为：

```
typedef Iterator<int> Int_iterator;
```

可以用 `Iterator<int>` 替换 `Int_iterator` 而得到 `sum` 函数和 `main` 程序：

```
int sum(Iterator<int> ir)  
{  
    int result = 0;  
    while (ir.valid( ))  
        result += ir.next( );  
    return result;  
}  
  
void main( )  
{  
    int numb[10];  
    for (int num=0; num<10; num++)
```

```

        numb[num] = num;
        cout<<sum(Iterator<int>(numb,10))<<"\n";
        return 0;
    }

```

既然能在任意类型的数组上进行迭代，它也能够求出任意类型的值的和。

将 sum 函数写为函数模板：

```

template <class T>
T sum(Iterator<T> ir)
{
    T result = 0;
    while (ir.valid( ))
        result += ir.next( );
    return result;
}

```

现在可以计算任意类对象数组的和，但要求这些类对象满足以下条件：

- ◎ 可以将 0 转换为这个类的对象；
 - ◎ 运算符 “+=” 在这个类对象中有定义；
 - ◎ 对象具有类似值（value_like）的语义，使得 sum 函数能够返回对象。
- 所有数值类型都满足这些要求。当然，其他类要这样做也是容易定义的。现在已经解决了三件事情中的两件，那么，第三件事呢？

3. 抽象出存储技术

我们正在寻找不同种类的迭代算子来反映不同的数据结构。至今仅有一个迭代算子，它让我们将元素存储在数组中。但是，如果元素存储在链表中呢？如果存储在文件中呢？抽象这些是可能的吗？

解决这个问题的“标准”方法是将 Iterator 类转换成一个抽象基类，它能描述大量不同迭代类中的任意一个：

```

template <class T>
class Iterator
{
public:
    virtual int valid( ) const = 0;
    virtual T next( ) = 0;
    virtual ~Iterator( ) { }
};

```

然后，上面所说的迭代算子 Array_iterator<T>是 Iterator<T>的一种：

```

template<class T>
class Array_iterator:public Iterator<T>
{

```



```
public:
    Array_iterator(T * p, int leng):data(p), len(leng)
    { }
    int valid( ) const
    {
        return len>0;
    }
    T next( )
    {
        --len;
        return *data++;
    }
private:
    T * data;
    int len;
};
```

同时，将 `sum` 函数的参数变元定义为 `Iterator` 的引用，以便允许动态匹配：

```
template<class T>
T sum(Iterator<T> & ir)
{
    T result = 0;
    while (ir.valid( ))
        result += ir.next( );
    return result;
}
```

这时，要累加一个数组元素的和，仅需要创建一个适当的 `Array_iterator` 来描述它，并将其传递给 `sum` 函数：

```
#include <iostream.h>
void main( )
{
    int numb[10];
    for (int num=0; num<10; num++)
        numb[num]=num;
    Array_iterator<int> it(numb,10);
    Iterator<int> I_it=&it;
    cout<<sum(I_it)<<"\n";
    return ;
}
```

这里的差别是，显式地创建了一个 `Array_iterator<int>` 对象 `it`，用它来表达在数组 `numb` 进行迭代。也可以这样表达：

```
cout<<sum(Array_iterator<int>(x,10))<<"\n";
```

由于 `sum` 函数的变元使用了动态匹配，`sum` 函数中的每个内循环都需要虚函数调用，这样开销太大了，特别对于像整型这样的简单类型来说，这个开销难以接受。

显然，可以采用另一种方法来解决，`sum` 用两个类型参数变元：迭代算子的类型以及要被相加的对象的类型。不幸的是，这又有麻烦了，如果将 `sum` 函数用下述方式定义为函数模板：

```
template <class Iter, class obj>
obj sum(Iter it)
{ ... }
```

将会发现，所定义函数的返回类型与其变元类型无关。也就是说，`sum(x)` 的返回类型 `obj` 与 `x` 的类型无关。这是不合法的。

因此，只有再次采用 C++ 语言的标准技术，使 `sum` 成为一个类。换句话说，不是定义 `sum` 为一个函数，而是定义一个 `sum` 类，它的目的是将迭代算子返回给它的值相加在一起。这表达起来很容易：

```
template<class S, class It>
class sum
{
public:
    sum(It iter):ir(iter)
    { }
    operator S( );
private:
    It ir;
};
```

提供了一个构造函数将 `It` 的对象转换为 `sum` 类型，一个类型转换函数 `operator S()` 将 `sum` 对象转换为 `S` 的类型，这样求和就容易多了：

```
template<class S, class It>
sum<S, It>::operator S()
{
    It iter=ir;
    S result=0;
    while (iter.valid())
        result += iter.next();
    return result;
}
```

可见，`operator S()` 与前面定义的 `sum` 函数没有太多的差别，因此，用起来也差不多。



至此，`sum` 类模板表达的概念是非常高级的，它有两个模板参数：相加的对象的类型（`class S`）和迭代算子的类型（`class It`）。模板参数 `class S` 表示 `sum` 能将任意类型 `S` 的对象进行相加，模板参数 `class It` 表示 `sum` 能用类属迭代算子 `It` 对任意数据结构进行迭代提取（或称遍历操作）。当创建一个 `sum` 对象时，首先通过构造函数将模板实参（一个迭代类对象）与形参 `iter` 相结合，对私有成员 `ir` 进行初始化。这意味着迭代算子的实例化。其次，隐含地使用类型转换函数 `operator S()`，对初始化了的 `ir` 进行迭代求和，并将结果值转换为 `S` 类型。整个求和过程是在创建一个 `sum` 对象时自动进行的。下面是 `sum` 类的使用方法：

```
#include <iostream.h>

void main()
{
    int numb[10];
    for (int num=0; num<10; num++)
        numb[num]=num;
    cout<<sum<int, Array_iterator<int>>>(Array_iterator<int> (numb,10));
    cout<<endl;
    return ;
}
```

`Array_iterator<int> (numb,10)` 创建一个 `Array_iterator<int>` 对象（迭代类对象），它描述了对数组 `x` 进行迭代。构造一个 `sum<int, Array_iterator<int>>` 的对象，它将数组中的元素求和，结果值为 `int` 类型。最后打印 `sum<int, Array_iterator<int>>` 的结果。

这种风格需要一些多余的代码，但换来了许多额外的灵活性。

还可以缩写成如下的普通的格式：

```
template<class T>
T sum(T * p, int n)
{
    return sum<T, Array_iterator<T>>> (Array_iterator<T>(p,n));
}
```

这样使用起来就容易多了：

```
void main( )
{
    int numb[10];
    for (int num=0; num<10; num++)
        numb[num]=num;
    cout<<sum(numb,10)<<"\n";
    return ;
}
```

以一个简单的 `sum` 函数为例，说明怎样将它分离成三部分，每部分都是独立的代码，并得到了很大的灵活性。随着模板技术的广泛采用，这种技术将改进我们的系统设计方法，特别是类库的设计。



10.10 堆栈、队列的应用

有这样一个停车场，如图 10-15 所示。它的停车规则如下：

- ① 车辆只能从入口进入，并且只能从出口驶出；
- ② 车辆进入后要排队等候进入车位；
- ③ 一个车位只能停一辆车；
- ④ 辅助车位不能停车，只用于调度。

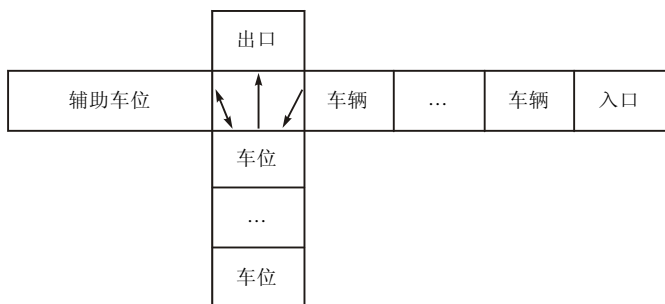


图 10-15 停车场示意图

根据这样的规则，可以制定下面的调度策略：

- ① 从入口进入的车辆在路上排队等候；
- ② 如果前面的车位有空，则等候的车辆可以依次驶入车位，后驶入的车辆停在前面的车位中；
- ③ 如果有车辆要从停车位出来，则停止进入，同时该车前面的所有车辆依次退入辅助车位，然后该车驶出，最后辅助车位中的所有车辆按进入的反序驶入停车位并占满后面的车位；
- ④ 重复第①步。

如果要用程序来模拟整个调度过程，那么首先要定义一些数据结构。很明显，等待车辆形成了一个队列，而车位和辅助车位都是堆栈。另外，由于车辆的到来、驶出都是随机发生的非顺序事件，所以车辆进出队列、堆栈的过程是并发的过程。这样，在并发过程中，必须注意临界资源（这里就是队列和堆栈）的共享互斥操作。

下面是整个模拟过程的全部代码。需要读者注意的是：代码中的并发函数是假设的。

```
class Stack                                //定义堆栈类
{
public:
    CriticalSection cs;                    //临界资源互斥信号
    //其他成员请参考本章的 Stack 类
};

class Queue                                //定义队列类，这个队列不是循环队列
```



```
{
public:
    CriticalSection cs;           //临界资源互斥信号
    //其他成员请参考本章的 Queue 类
};

typedef int CarID;
Stack      CarPark;             //停车位堆栈
Stack      AuxCarPark;          //辅助车位堆栈
Queue      CarWait;             //等待队列
Semaphore  sCarComing;          //信号量，指示有车辆到来
Semaphore  sVacant;             //信号量，指示有空停车位
Semaphore  sCarOut;             //信号量，指示有车辆出去
//线程模板，模拟车辆到来
void WaitCar()
{
    CarID Car;
    while (true)                //24 小时服务
    {
        ::Reset(sCarComing);    //复位信号量
        ::Wait(sCarComing);      //线程阻塞在这里，直到信号量被置位
        Car = ::GetCarID();      //调用全局函数获得车辆号
        CarWait.cs.Lock();       //队列加锁，阻塞其他线程访问
        if (CarWait.IsFull())
            ::Reject(Car);       //队列满，请车辆到别的停车场
        else
            CarWait.Append(Car);
        CarWait.cs.Unlock();     //解锁
    }
}
//线程模板，模拟车辆进入车位
void Park()
{
    CarID Car;
    while (true)                //24 小时服务
    {
        ::Reset(sVacant);       //复位信号量
        ::Wait(sVacant);        //线程阻塞在这里，直到信号量被置位
        CarWait.cs.Lock();
    }
}
```

```

        CarWait.Get(Car);           //排在最前面的车辆出队
        CarPark.cs.Lock();
        CarPark.Push(Car);         //车辆驶入最前面的停车位
        CarPark.cs.Unlock();
        CarWait.cs.Unlock();
    }
}
//线程模板，模拟车辆驶出
void CarOut()
{
    CarID CarOut;
    CarID Car;
    while (true)
    {
        ::Reset(sCarOut);           //复位信号量
        ::Wait(sCarOut);             //线程阻塞在这里，直到信号量被置位
        CarOut = ::GetCarID();       //获得要驶出的车辆号
        CarPark.cs.Lock();
        AuxCarPark.cs.Lock();
        while (true)                 //该车辆前面所有的车依次进入辅助车位
        {
            Car = CarPark.Pop();
            if (Car == CarOut) break;
            AuxCarPark.Push(Car);
        }
        ::CarGoingOut(CarOut);       //该车驶出
        while (!AuxCarPark.IsEmpty()) //辅助车位里的车辆依次进入车位
        {
            Car = AuxCarPark.Pop();
            CarPark.Push(Car);
        }
        AuxCarPark.cs.Unlock();
        CarPark.cs.Unlock();
        Signal(sVacant);             //发出有空位信号
    }
}
void main()
{
    BeginThread(WaitCar);            //启动线程

```



```
BeginThread(Park);
BeginThread(CarOut);
bool bStop = false;
while (!bStop)
{
    switch (getch())
    {
        case 'c':
            Signal(sCarComing);           //模拟有车前来停泊
            break;
        case 'o':
            Signal(sCarOut);               //模拟有车开出
            break;
        case 'q':                           //结束
            bStop = true;
    }
    KillThread(WaitCar);                  //杀掉所有线程
    KillThread(Park);
    KillThread(CarOut);
}
```

习题 10

10.1 声明一个哺乳动物类 **Mammal**，再派生出一个 **Cat** 类，声明一个 **Cat** 类的对象，观察基类与派生类的构造函数和析构函数的调用顺序。

10.2 设计一个圆类 **Circle** 和一个桌子类 **Table**，另设计一个圆桌类 **Roundtable**，它是从前两个类派生的，要求输出一个圆桌的高度、面积和颜色等数据。

10.3 编写一个程序设计一个汽车类 **Vehicle**，包含的数据成员有车轮个数 **Wheels** 和车重 **Weight**，可以显示这些信息，可以获取对象的车轮个数和每个车轮的承重。

(1) 小车类 **Car** 是它的私有派生类，其中包含载人数 **Passenger_load**。可以获取载人数和显示相关信息。

(2) 卡车类 **Truck** 是 **Vehicle** 的私有派生类，其中包含载人数 **Passenger_load** 和载重量 **Payload**，可以获得载人数和载货物的效率，显示相关信息。

(3) 测试上述功能。

10.4 用面向对象方法实现异质链表。所谓异质，是指链表中各表项内容的类型不要求相同。

(1) 以大学环境为例，这里包括学生、职员和教师。希望对这些人的信息进行管理。

① 学生：姓名、年龄、身份证号码、年级和平均成绩。

② 职员：姓名、年龄、身份证号码和小时工资。

③ 教师：姓名、年龄、身份证号码和年工资。

要求能实现以下三种操作：

① 插入，向异质链表中增加一个学生、职员或教师的信息。

② 删除，从链表中删除一个学生、职员和教师的信息。

③ 打印，显示链表中所有的信息。

(2) 对于一些既是学生又是职员双重身份的人，应能单独进行记录。

10.5 定义两个类：

```
class Fruit
{
public:
    virtual char * identify() {return "Fruit"; }
};

class Tree
{
public:
    virtual char * identify() {return "Tree"; }
};
```

请派生出一些既是 Fruit 又是 Tree 的类及其派生类。例如，定义 Apple 类：

```
class Apple: public Fruit, public Tree
{
...
};
```

每个派生类至少有一个成员函数显示自己的类名，以及从哪些基类中派生出来，例如，定义 Apple 类应该能显示如下信息：

(Apple : Fruit, Tree)

这样，Apple_Pear（苹果梨）类应该显示的信息为

(Apple_Pear : (Apple : Fruit, Tree), (Pear : Fruit, Tree))

10.6 有 5 种图形：圆 (Circle)，正方形 (Square)，矩形 (Rectangle)，梯形 (Trapezoid)，三角形 (Triangle)，要求具备求图形面积的功能。在这些图形的基础上抽象出抽象基类 Shape，并使用基类指针数组，使它的每一个元素指向一种派生类对象。

第 11 章 Java语言基础



11.1 Java语言的发展历程

Java 语言是一种新型的程序设计语言，广泛地应用于 Internet 网络程序设计，它是跨平台的适用于分布式计算机环境的面向对象的程序设计语言。其“编写一次，到处运行”的跨平台优势给整个网络世界带来了巨大变革，为软件开发者提供了充分展示的舞台。

这一切都起源于 20 世纪 90 年代初期的绿色计划（Green Project）和 Oak 语言（Java 语言的雏形）。1990 年 12 月，Sun 公司的绿色计划启动，由 James Gosling 牵头，有 13 人参加，绿色计划不仅创建了 Oak 语言，同时还创建了一种操作系统、一种图形用户界面和一种名为 Star7（*7）的手持设备。

1991 年 4 月，James Gosling，以 C++ 语言为基础，开发了一种可在不同平台上工作、可令不同设备互连的程序设计语言——Oak 语言。为了证明这种新语言对数字设备的未来具有多么重大的影响，绿色团队瞄准数字有线电视业开发了一种交互、手持式的家庭娱乐设备控制器。但是，这种理念在当时显得太超前了，数字有线电视业对于 Java 技术能带给他们的飞跃还没有做好充分的准备。

1994 年，Web 大发展，Sun 公司的共同创始人 Bill Joy（Berkeley UNIX 的创始者）坚信 Oak 是唯一可以令 Web 真正实现交互的途径。在他的推动下，Oak 变成了用来创建一种叫 Applet（小程序）的语言，这些 Applet 可在任何操作系统中运行。

1995 年 1 月，Sun 公司发现 Oak 名字已经被别人注册，因此更名为 Java。绿色团队还开发了第一个支持 Java Applet 的 Web 浏览器——Hotjava。

1995 年 5 月，Sun 公司在 San Francisco 举行的 Sunworld 会议上正式公布了 Java 技术。在此次会议上，Netscape 公司宣布将在其 Web 浏览器产品中支持 Java。同年冬天，Navigator 2.0 正式支持 Java。

不久，Sun、SGI 和 Macromedia 三家公司宣布联合制定基于 Java 的开放式多媒体标准。随后许多著名大公司，如 IBM、Microsoft、Novell、Oracle、Borland 等，都相继宣布支持 Java。

1996 年 1 月，第一个 Java 开发工具包（Java Development Kit）：JDK1.0 诞生。1996 年 4 月，10 个最主要的操作系统供应商表明将在其产品中嵌入 Java 技术；1996 年 9 月，约 8.3 万个网页应用了 Java 制作技术。

1997 年 2 月 18 日，JDK1.1 发布。1997 年 4 月 2 日，JavaOne 会议召开，参与者逾 1 万人，创当时全球同类会议规模之纪录。1997 年 9 月，Java Developer Connection 社区成员超过 10 万。1998 年 2 月，JDK1.1 被下载超过 200 万次。

1998 年 12 月 4 日, JDK1.2 隆重发布, 标志着 Java2 平台的诞生。

1998 年 12 月 8 日, Java2 企业级平台 J2EE 发布。

1999 年 6 月, SUN 公司发布 Java 的 3 个版本: 标准版、企业版和微型版 (J2SE、J2EE 和 J2ME)。

◎ J2SE (Java 2 Standard Edition): 它是一组针对传统桌面应用的 API 和运行环境。

◎ J2EE (Java 2 Enterprise Edition): 它是 J2SE 的扩展集, 主要用于在服务器端开发可伸缩、可迁移、以数据库为核心的企业级应用。

◎ J2ME (Java 2 Micro Edition): 它定义了一组针对嵌入式设备和消费电子设备的 API 和运行环境, 这些设备包括无线手持设备、PDA、电视机顶盒以及其他缺乏足够的资源去支持 J2SE 的设备。

把 Java 2 平台分成三种版本使得它能够更好地满足不同目标领域中开发者的需求, 同时也使 Java 技术在保持其“编写一次, 到处运行”精神的同时, 在不同领域得到继续发展。

2000 年 5 月 8 日, JDK1.3 发布。

2001 年 6 月 5 日, NOKIA 公司宣布, 到 2003 年将出售 1 亿部支持 Java 的手机。

2001 年 9 月 24 日, J2EE1.3 发布。

2002 年 2 月 13 日, JDK1.4 发布。

2002 年 2 月 26 日, J2SE1.4 发布, 自此 Java 的计算能力有了大幅提升。

2004 年 9 月 30 日, J2SE1.5 发布, 是 Java 语言发展史上的又一里程碑事件。为了表示这个版本的重要性, J2SE1.5 更名为 J2SE5.0。

2005 年 6 月, JavaOne 大会召开, Sun 公司公开 Java SE 6。此时, Java 的各种版本已经更名以取消其中的数字“2”: J2EE 更名为 Java EE, J2SE 更名为 Java SE, J2ME 更名为 Java ME。

现在, 在它诞生后的 10 年, Java 平台已经吸引了 400 多万软件开发商, 全世界的各个行业都在使用它, 任何使用编程技术的设备、计算机和网络都在大范围地应用它。

事实上, Java 技术的多功能性、有效性、平台的可移植性以及安全性已经使它成为网络计算领域最完美的技术。到今天为止, Java 技术为 25 亿台设备提供支持, 包括:

◎ 7 亿台以上的 PC;

◎ 7 亿 8 百万部移动电话以及其他手持式设备;

◎ 10 亿个智能卡以及机顶盒、打印机、网络照相机、游戏、汽车导航系统、彩票终端、医疗设备、收费站等。

今天, 无论是互联网和科学超级计算机还是膝上型计算机和手机, 无论是华尔街的市场模拟器还是家庭游戏机和信用卡, 在所有网络和设备上都会看到 Java 技术的身影, 它已经无处不在。日臻完善、极度强大而且功能繁多的 Java 技术已经成了开发商的无价之宝, 利用它可以:

◎ 在一个平台上编写软件, 然后在另一个平台上运行;

◎ 创建可在 Web 浏览器和 Web 服务中运行的程序;

◎ 开发适用于联机论坛、存储、投票、HTML 格式处理以及其他用途的服务器端应用



程序;

- ◎ 将基于 Java 技术的应用程序或服务组合在一起,以生成高度自定义的应用程序或服务;
- ◎ 为移动电话、远程处理器、低成本的消费产品以及任何具有数字核心的设备编写强大而高效的应用程序。



11.2 Java语言的特点

Java 并不仅仅是一种计算机语言,Java 实际上也是一个紧凑、健壮、安全、跨平台和基于网络的计算环境,Java 计算的概念,是一种 C/S、B/S 等结构的解决方案。Java 主要有两个优势,一是标准性和连接性,二是显著加快应用程序开发。由此加快了计算机化的步伐,特别是 Web 技术的推广应用。

Java 语言是一种新型的编程设计语言,广泛地应用于 Internet 网络编程设计。它是跨平台的适用于分布式计算机环境的面向对象程序设计语言。它具有简捷、安全、面向对象、动态、体系结构中立、可移植、高性能、多线程、解释执行、分布式等特性。



11.2.1 简捷性

Java 语言由 C++语言衍生而来,其语言风格与 C++语言十分类似,但进行了很大的简化和改进,例如,C++语言中的指针和多重继承常常使程序复杂化。而 Java 语言不支持 C++语言的内存单元指针,通过符号指针来引用内存,而符号指针由 Java 运行系统在运行时具体解释为实际的内存地址;Java 只支持单重继承,但支持接口 (Interface),一个类可以实现多个接口,利用接口可以得到多继承的优点,又没有多继承混乱、复杂的问题。

Java 语言采用自动内存分配和回收大大简化了程序设计者的内存管理工作,而 C++语言要求程序员对内存进行分配和回收。对于像 Java 这样的功能强大的语言而言,清晰的语法使得 Java 程序容易编写和阅读。Java 语言采用面向对象的编程在语言结构上是简捷的,另一方面它可以使编程者能够尽量避免在编程中出现错误。



11.2.2 面向对象

Java 语言具有真正的面向对象语言的特点,它完全基于类、对象,以类的形式组织代码、数据类型。它支持静态和动态的代码继承及重用,也具有面向对象所共有的特性:封装、继承、多态性。

所谓面向对象是一种计算机编程方法,它是对具有相同属性和行为的现实世界中的对象抽象,并映射到计算机的编程上,用数据表示属性,程序执行代码表示行为。

图 11-1 表示了类、对象、实体的相互关系和面向对象的问题求解的思维方式。在用面向对象的软件方法解决现实世界问题时,首先将物理存在的实体抽象成概念世界的抽象数据类型,这个抽象数据类型里面包括了实体中与需要解决的问题相关的数据和属性;然后再用

面向对象的工具, 如 Java 语言, 将这个抽象数据类型用计算机逻辑表达出来, 即构造计算机能理解和处理的类; 最后将类实例化就得到现实世界实体的面向对象的映射——对象, 在程序中对对象进行操作, 就可以模拟现实世界中的实体上的问题并解决之。

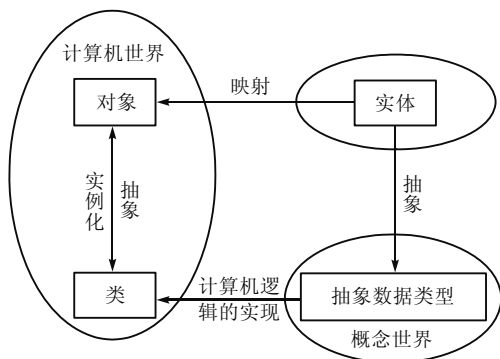


图 11-1 类、对象与实体

面向对象程序设计语言的任何方面均涉及对消息或对象的处理, 对象间所有可能的互操作都是通过消息传递来实现的。面向对象编程就是对对象和消息编程, 它支持封装、多态性和继承。封装就是将对象内的数据和代码联编起来, 形成一个对象; 多态性是指, 一个接口有多个内在实现形式表示; 继承是指, 某一类对象直接使用另一类对象的所有属性和行为的过程。它可以简化类的定义, 即在定义某个类时, 可以只定义有别于父类的属性或行为, 而其他部分则可以从父类那里继承来重用。

图 11-2 举例表明了如何采用面向对象的方式解决现实世界的问题。每个类有一些方法（函数）对应可能收到的请求, 对某个对象发出某个请求（消息）, 某个方法就被调用, 进而执行对应的程序代码。下面我们以灯的行为为例来进行说明。我们为“灯”定义一个类 `Light`, 然后创建一个对象 `lt`。能够对 `Light` 的对象发出的请求是: 打开、关闭、变亮、变暗。图 11-2 中的第一条语句, 创建了 `Light` 的对象 `lt`, 用 `new` 运算符给 `lt` 开辟内存空间, 并调用构造函数 `Light()` 对对象 `lt` 进行初始化, 这实际上就是实例化一个对象。第二条语句, 就是调用对象 `lt` 的 `on()` 方法, 使 `lt` 对应的灯变亮。

图 11-3 举例说明了面向对象的继承和多态性。用 Java 语言来编写方法如下（你很快就会学到如何编写 Java 程序了）:

错误!



图 11-2 类的实例化

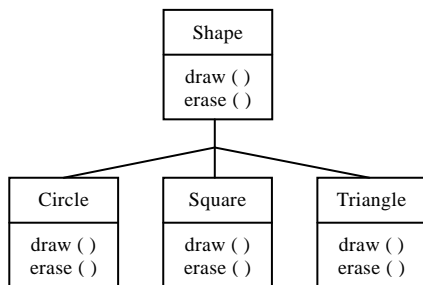


图 11-3 继承与多态性



```
void doStuff (Shape s) {  
    s.erase ();  
    //...  
    s.draw ();  
}
```

上面的方法可以和任何 **Shape** 类（其他几何形状的父亲）传递消息，独立于它所绘制或擦拭的任何特定子对象类别（如圆、线、矩形）。如果我们在程序的其他地方用到了 **doStuff()** 方法：

```
Circle c = new Circle();  
Triangle t = new Triangle();  
Square s = new Square();  
doStuff(c);  
doStuff(t);  
doStuff(s);
```

那么，当调用 **doStuff()** 时，不论对象的实际类型是什么，都能运作无误。我们以 **doStuff(c)** 语句为例来看看是如何运作的。当 **Circle** 的对象 **c** 被传入这个本来接收 **Shape** 对象的方法时，因为 **c** 是 **Shape** 类的子类的对象，也是一种特殊的 **Shape** 对象，所以它也被 **doStuff()** 认可。我们把这种“将子类视为其父类”的现象，称为“向上转型(Upcasting)”。接着来看看 **doStuff()** 内的程序代码：

```
s.erase ();  
//...  
s.draw ();
```

注意：这里并不像面向过程编程那样：“如果你是 **Circle**，请你做这些；如果你是 **Square**，请你做这些；如果你是 **Triangle**，请你做这些；……”如果编写这样的代码，每当有新的 **Shape** 子类加入时，都得修改这段程序代码。这里所表现的意思是：“如果你是相应的 **Shape** 子类，你就知道调用相应的子类的 **erase()** 方法和 **draw()** 方法。”调用 **Circle** 的 **draw()** 方法与调用 **Triangle** 的 **draw()** 方法所执行的代码完全不同。

当编译器编译 **doStuff()** 方法时，无法精确知道 **doStuff()** 所处理的实际类型，所以它调用父类 **Shape** 的 **erase()** 方法和 **draw()** 方法；但执行的时候可以确定是哪个 **Shape** 的子类，调用正确的 **erase()** 方法和 **draw()** 方法。这就是多态性的神奇之处，它能轻松地实现“动态绑定（后期绑定）”。



11.2.3 动态性

Java 语言能够适应发展变化的环境。由于 Java 语言编程的基本组成单元是类，在运行中，Java 的类是动态装载的，因此 Java 能在分布式系统中动态地维护应用程序，并支持类库间一致性，避免诸如类库升级后需要重新编译整个应用程序这一类问题。

Java 的设计使它能够适合一个不断发展的环境,在类库中可以自由地加入新的方法和实例变量而不会影响用户程序的执行,并且 Java 通过接口来支持多重继承,使之比多重继承具有更灵活的方式和扩展性。

Java 语言滞后联编机制充分利用面向对象编程风格的优点,真正做到即插即用的模块功能。

11.2.4 安全性

Java 虚拟机对字节码进行加密传输、客户端校验以及对解释器在客户端临时分配、布置内存管理来保障其安全性。

Java 语言对内存访问都是通过对对象实例变量实现的,以防止用户在网络系统或分布系统环境下使用特洛伊木马等手段访问对象的私有成员。Java 语言不支持指针,从而也避免了指针操作的安全隐患,Java 语言提供的内存管理机制,有自动搜集“内存垃圾”程序。Java 在字节码的传输过程中使用了公开密钥加密机制(PKC),在运行环境中提供了 4 级安全性保障机制:

- ◎ 字节码校验器(ByteCode Verifier);
- ◎ 类装载器(Class Loader);
- ◎ 运行时内存布局;
- ◎ 文件访问限制。

11.2.5 平台无关性和可移植性

用 Java 语言编写的程序,不经任何改动就可以在不同的硬件或软件平台上执行,这是因为 Java 编译器所生成的可执行代码是基于一种抽象的处理器——Java 虚拟机(Java Virtual Machine, JVM)来实现的。Java 虚拟机就是虚拟运行 Java 代码的假想计算机,它可以定义为:运行经过编译的 Java 目标代码的计算机的实现。

Java 程序运行前,首先要经过编译,再进行解释。Java 编译器所生成的代码不针对任何具体的硬件体系结构和软件平台,这种代码被称为“字节码”,它与硬件体系结构和软件平台无关。字节码在运行过程中,是由针对于运行系统硬件体系结构和软件平台的 Java 解释器,将其转换成该系统对应的指令代码而实现运行的。

Java 体系结构的中立性,能够保证字节码文件可以在任何支持 Java 的平台上运行,也就是说,Java 程序不需要重新编译就能在任何平台上运行,因此它具有很好的可移植性。

Java 编译器将 Java 的源程序编译成 JVM 可执行的代码——字节码,而不像 C 语言和 C++ 语言编译器,生成直接能运行于某种特定硬件平台的可执行代码。后者,在编译过程中就确定了内存分配情况;而前者,是由解释器在运行过程中创立内存布局的,这样,更加有效地保证了 Java 的可移植性和安全性。

11.2.6 高性能

虽然 Java 是解释执行语言,但它的字节码在编译生成时,带有许多的编译信息,在 Java



字节码格式设计中充分考虑到它的机器码执行效率,很容易直接转换成对应于特定处理器的高性能机器码。因此 Java 字节码的执行效率非常接近于 C 语言或 C++语言生成的机器码执行效率。

- ◎ 解释执行方式: 由解释器通过每次翻译并执行一小段代码来完成字节码程序的所有操作, 执行效率接近于 C 语言或 C++语言生成的机器码执行效率。
- ◎ 即时编译方式: 由代码生成器先将字节码转换成机器码, 再全速执行该机器码, 执行效率可以与 C 语言或 C++语言生成的机器码执行效率媲美。
- ◎ 字节码在 JVM 中运行, 分为以下 3 个阶段:
- ◎ 代码的装入, 是由类装载器 (Class Loader) 完成;
- ◎ 代码的校验, 用于发现各种可能出现的错误;

◎ 代码的运行, 在代码校验后就可以执行了。

图 11-4 说明了目前 Java 代码的两种执行方式。

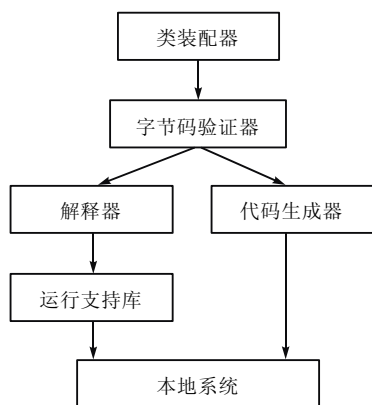


图 11-4 Java 运行系统

11.2.7 多线程

多线程机制使应用程序能够并行执行, 而且同步机制保证了对共享数据的正确操作。通过使用多线程, 程序设计者可以分别用不同的线程完成特定的行为, 而不需要采用全局的事件循环机制, 这样就很容易实现网络上的实时交互操作。

Java 多线程支持表现在两个方面: 一方面它自身的多线程, 可以利用系统的空闲执行一些常规处理等; 另一方面,

它提供对多线程的语言级支持, 提高程序执行效率。

11.2.8 分布式

Java 是一个适用于网络的语言, 它的设计使分布式计算变得容易起来。它提供的类库支持对 TCP/IP 协议处理, 可以通过 URL 地址访问网络上其他的对象。它支持 WWW 的 C/S 和 B/S 的计算机网络模型, 支持分布式的数据分布和操作分布。它是一门非常适合 Internet 网络和分布式环境的编程语言。

11.2.9 健壮性

Java 在编译源程序及运行程序中, 对源程序中的数据和标识符等, 要进行数据类型的匹配, 数组下标越界检查, 避免由这些错误引发的系统问题。

Java 不支持指针, 并采取的保护内存数据、程序的机制以及自动内存管理, 碎片收集为系统的健壮性提供了保障。

Java 提供的异常控制机制, 能够正确地处理在运行时所出现的错误, 使应用程序在运行中, 能够更好地适应各种情况的出现。



11.3 Java语言的开发工具包

学习 Java 需要一个程序开发环境，目前有很多很好的 Java 图形化集成开发环境（IDE）可用，包括来自 Sun、Borland、IBM 等公司的产品，其中开放源码的免费集成开发环境有：Sun 公司的 NetBeans（目前最新版本 5.0），IBM 公司的 Eclipse；商业集成开发环境有：Borland 公司的 JBuilder（目前最新版本 JBuilder 2005），IBM 公司的 WebSphere 系列，Sun 公司的 Java Studio 系列。

但是，从初学者角度来看，采用 Sun 公司提供的免费 Java 开发工具包 JDK（Java Development Kit）来开发 Java 程序，能够很快理解程序中各部分代码之间的关系，有利于理解 Java 面向对象的程序设计思想。JDK 的另一个显著特点是随着 Java（J2EE、J2SE 以及 J2ME）版本的升级而升级。

JDK 简单易学，可以通过任何文本编辑器（如 Windows 的记事本、UltrEdit、JavaFree 等）编写 Java 源文件，然后在 DOS 模式下编译和运行程序，使初学者能把精力先集中在学习 Java 程序本身，而不是迷失在 Java 图形化集成开发环境的复杂关系之中。

11.3.1 JDK 的下载、安装和设置

1. JDK 的下载和安装

可以到 Sun 的公司的网站：<http://java.sun.com/javase/downloads/index.jsp>，免费下载目前 JDK 的最新版本 JDK 6.0。

注意：JDK 中包含了 Java 运行环境（Java Runtime Environment，JRE），而 JRE 是针对特定平台的，所以下载 JDK 的时候，要看清楚它是针对哪个平台的，是 Windows 还是 Linux 或其他平台。例如，针对 Windows 平台的安装包名为：jdk-6u6-windows-i586-p.exe。

运行下载的 JDK 安装包就可以在机器上安装 Java 开发环境了，假设安装到 C:\Program Files\Java 目录下，则会生成如图 11-5 所示的目录结构。安装完之后，就可以编写 Java 程序，并进行编译和运行了。

2. PATH 参数的设置

在 JDK 的安装目录的 jdk1.6.0_06\bin 子目录下含有编译器（javac.exe）、解释器（java.exe）等可执行文件。为了编程的方便，还需要配置操作系统的环境变量 PATH 参数，即把 bin 目录包含在 PATH 设置中。

对于 Windows 2000 或 XP，用鼠标右键单击“我的电脑”图标，弹出快捷菜单，然后选择“属性”命令，弹出“系统属性”对话框。单击该对话框中的“高级”选项卡，然后单击“环境变量”按钮，就会弹出“环境变量”对话框。

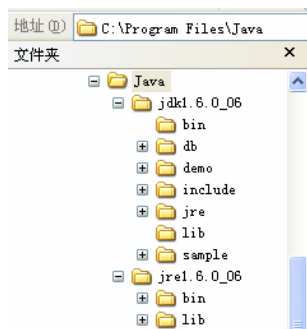


图 11-5 JDK 6.0 目录结构



如果环境变量中没有包含 **PATH** 参数,则单击“新建”按钮,就会弹出“新建用户变量”对话框,如图 11-6 所示,填写相应文本框就可以了。

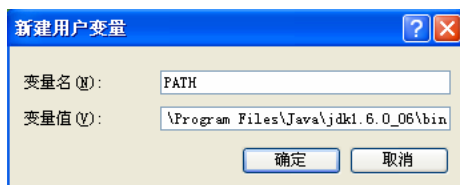


图 11-6 PATH 参数设置

如果环境变量中已经包含 **PATH** 参数,则直接双击 **PATH** 参数,就会弹出“编辑用户变量”对话框,在“变量值”文本框已有的内容后面加上

“; C:\\Program Files\\Java\\jdk1.6.0_06\\bin”即可。

3. CLASSPATH参数的设置

CLASSPATH 环境变量用于指示 **JRE** 搜索要运行的类文件的路径。在 **JDK** 的安装目录的 **jre1.6.0_06** 子目录下包含了 **Java** 运行程序时所需要的 **Java** 核心类库和虚拟机,这些类库被包含在 **jre1.6.0_06\\lib** 子目录下的压缩文件 **rt.jar** 中。

安装 **JDK** 一般不需要设置环境变量中的 **CLASSPATH** 的值,但如果你的机器中安装过一些商业化的 **Java** 开发工具或带有 **Java** 技术的产品,如 **PB**、**Oracle** 等,这些产品安装后,可能会修改 **CLASSPATH** 的值,那么当运行 **Java** 应用程序时,系统可能会加载这些产品所带的老版本的类库,从而导致程序要加载的类无法找到,使程序运行出现错误。例如, **CLASSPATH** 的值被设置为:

```
d:\\PB\\jdk1.1.7\\jre\\lib\\classes.zip;.
```

注意: **CLASSPATH** 值中的“.”表示当前目录,如果有多个目录需要加入环境变量的值中,则用“;”进行分割。

如果不想删除这些产品的 **CLASSPATH** 设置,可以重新编辑环境变量 **CLASSPATH** 的值,把 **rt.jar** 所在的子目录加进去:

```
C:\\Program Files\\Java\\jre1.6.0_06\\lib\\rt.jar;
```

11.3.2 JDK的简介

针对各个版本 **JDK** 最全面、权威的详细文档,就是 **Sun** 公司的 **Java Documentation** 文档,例如,针对版本 6 的文档名为: **Java(TM) SE Development Kit Documentation 6** 类库文档,对应的压缩文件名为 **jdk-6-doc.zip**。可以到 **Sun** 的公司的网站:免费下载 **JDK 6.0 Documentation**。

JDK 6.0 体系结构如图 11-7 所示,自上到下,该体系结构分为两大层。

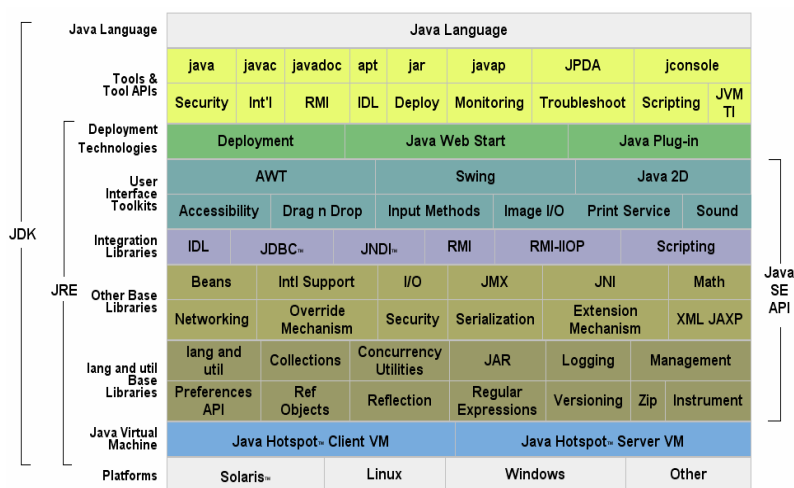


图 11-7 JDK 6.0 体系结构

◎ **JDK**：包括 Java 的开发环境（编译、调试等工具）和运行环境。

◎ **操作系统平台**：包括 Sun 公司的 Solaris、Microsoft 公司的 Windows 系列，以及 Linux 和其他操作系统，如 IBM AIX、HP-Unix 等。

JDK 又分为三部分。

◎ **Java 语言**：面向对象的程序设计语言，所编写的程序经过编译以后生成能被 JVM 识别的字节码。

◎ **Java 开发工具和应用编程接口（Tools and Tools API）**。

◎ **Java 运行环境 JRE**。

Java 运行环境由三大层组成：

◎ **部署技术（Deployment Technologies）**；

◎ **Java SE 应用编程接口（Java SE API）**；

◎ **Java 虚拟机（Java Virtual Machine, JVM）**。

Java SE 应用编程接口由 4 层组成：

◎ **用户接口工具包（User Interface Toolkits）**；

◎ **集成库（Integration Libraries）**；

◎ **其他基础类库（Other Base Libraries）**；

◎ **语言和实用工具基础类库（Lang&Util Base Libraries）**。

对于学习 Java 程序设计的初学者而言，一般只需了解和掌握常用命令，本节我们介绍 4 个常用命令。

◎ **javac**：Java 编译器。

◎ **java**：Java 解释器。

◎ **appletviewer**：下载并运行 HTML 文件中的 Applet 程序。

◎ **javadoc**：Java 文档生成器。



1. javac

Java 编译器 `javac` 本身就是一个用 Java 语言编写的应用程序，它与其他编译语言不同，它的作用是把 Java 源文件（扩展名为 `.java` 文件）编译成类文件，即 Java 字节码文件（扩展名为 `.class` 文件）。这种字节代码与机器代码类似，却不针对具体的机器，由 JVM 装载和执行。

`javac` 的执行方式如下：

```
javac [<options>] <source_file>
```

说明如下。

① `<options>` 为编译选项，它可以包含许多选项，初学者只要掌握常用选项就可以了。

`-classpath <路径; 路径; ...>`：指定 `javac` 在编译过程中，要引用类的搜索路径表，路径间以 “;” 分隔。

`-d <目录; 目录; ...>`：指定生成的类文件的存放路径，目录之间用 “;” 分隔。注意，它要与 Java 程序中的 `package` 联合使用。如果没有 `-d` 选项，则生成的类文件与源文件在同一个目录下。

② `<source_file>` 是扩展名为 `.java` 的源文件，可以是一个源文件，或多个源文件的列表。

2. java

编译后的字节码是不能直接在机器上运行的，要通过 Java 解释器 `java` 解释执行。其调用格式为：

```
java [<options>] <classname> [<arguments>]
```

说明如下。

① `<options>` 包含许多选项，初学者只要掌握常用选项

`-classpath <路径; 路径; ...>`：指定要运行的类文件的目录。如果有多个，路径间以 “;” 分隔。

② `<classname>` 是扩展名为 `.class` 的类名，即要执行的程序，该类中必须包含一个 `main()` 方法。程序的执行是从 `main()` 开始的。`main()` 方法的格式声明如下：

```
public static void main(String args[])
```

③ `<arguments>` 部分，输入保存在 `main()` 方法中的 `args[]` 数组中的参数。

Java 解释器实质上是字节码解释器，指定一个类文件名，它会自动装载程序中需要的类文件，通过检查机制确定一个类是否合法，确保解释执行的字节码不会破坏 Java 语言的约定。

3. appletviewer

Applet 是用 Java 语言编写的一类小程序，它能在小应用程序观察器 `appletviewer` 或在支持 Java 的 Web 浏览器上执行。小应用程序观察器的作用是下载 HTML 文档中的 Applet，其调用格式为：

```
appletviewer [-debug] urls
```

说明如下。

① `-debug` 为可选项，它的作用是小应用程序观察器将由 `JDb` 内部启动，可以调试被 HTML 文档中所引用的 Applet。

② `urls` 是统一资源定位符，是 Internet 网上资源的名称和地址标识。如果执行当前目录下的 HTML 文件，可以直接使用文件名。

4. javadoc

`javadoc` 是一个文档生成工具，它会分析 Java 源文件，并提取采用 “`/**注释内容*/`” 形式的注释，然后生成十分专业的描述类、接口、构造器、方法和属性的 HTML 格式的程序文档。

在开发大型应用时，维护和管理程序文档是非常重要且繁杂的工作，`javadoc` 工具能提供很好的支持，简化程序文档的生成和管理。由于本书重点不在这里，所以只是提醒大家有这样的工具，不具体介绍其用法，如果有兴趣，可以查看相关文档。



11.4 Java程序的基本结构

Java 语言的源程序是以 `.java` 为扩展名的文件，这些文件就是 Java 编译器 `java` 的编译单元。而每个单元又由 `package` 语句、`import` 语句、类 `class` 声明或接口 `interface` 声明语句构成。

① 包 (`package`) 是类和接口的集合，即类库。Java 语言用类库管理类，能够方便管理，减少类名间的竞争。Java 的类都包含在类库中，`package` 语句可用来指定类所属的类库。

② `import` 语句类似 C 或 C++ 语言中的 `include` 语句，为程序装载类或包，使程序能够使用 Java 环境下的其他类。

③ 类 (`class`) 声明语句用来声明类的名字及相关属性等内容。

④ 接口 (`interface`) 声明语句用来声明接口的名字及其相关属性。

示例代码如下，存入文件 `ClassName.java` 中：

```
package Name_of_Package;
import OtherClassName;

class ClassName {
    public static void main(String args[]){
        ...
    }
}
```

程序是由类组成的。在一个 Java 源程序中可以有多个类，用 `class` 关键字来创建类，在上例中，`class ClassName` 语句声明了一个名为 `ClassName` 的类。

这里 `package` 语句将正在创建的类 `ClassName` 放到包 `Name_of_Package`（即类库）中，一个包名等同一个文件系统目录名。

`import` 语句装载了一个名为 `OtherClassName` 的类，在源程序中，可以由 `import` 语句引入需要的类和接口。

`class` 声明语句则声明了一个类名为 `ClassName` 的类及其相关属性。

`public static void main(String args[])` 是 `ClassName` 类的 `main()` 方法的标识，其中 `main()` 方法的标记有三个修饰符，其含义分别如下。

◎ `public`：表示 `main()` 方法能被任何对象调用。



◎ **static**: 表示 `main()` 方法是一个类的方法。

◎ **void**: 表示 `main()` 方法不返回任何值。

另外, `main()` 方法的参数 `args[]` 是一个 `String` 类型的数组。

有关 `main()` 方法的具体内容, 留在后面的章节中讨论。



11.5 Java程序开发实例

Java 程序分为两大类型:

① 一类是由 Java 解释器控制执行的 Java Application, 它可以在任何装有 JVM 的计算机上运行;

② 另一类是嵌入到 Web 页面中, 由 Java 兼容浏览器控制执行的 Java Applet, 它采用一种“寄生”运行方式, 它依赖于 HTML 文件以及 Web 浏览器。



11.5.1 一个简单的Java Application程序

Java Application 是可独立运行的 Java 程序, 它由一个或多个类组成, 其中必须有一个类中定义了 `main()` 方法。`main()` 方法就像 C 语言的 `main` 函数一样, 是 Java Application 运行的起始点。

要实现 Application 程序从编写到运行的目的, 需要按下列步骤进行。

1. 首先创建一个Java的Application源程序 (.java文件)

要创建一个名为 `HelloWorldApp.java` 的文件, 可在任何字符编辑器中输入下列 Java 源程序代码:

```
class HelloWorldApp
{
    public static void main(String args[])
    {
        System.out.println("Hello, Java World !");
    } //end of main method
} //end of class
```

上述代码的实质是创建一个名为 `HelloWorldApp` 类, 并把它保存在与它相同名字的文件中 (即 `HelloWorldApp.java` 文件)。

注意: `main()` 方法是一个特殊方法, 所以它的方法头必须按下面的格式书写:

```
public static void main(String args[])
```

上例中的 `main` 方法中只有一条语句:

```
System.out.println("Hello, Java World !");
```

这条语句把字符串“Hello, Java World !”输出到系统的标准输出上, 如系统屏幕。其中,

`System` 是系统类的对象；`out` 是 `System` 对象中的一个域，也是一个对象，表示“标准输出”；`println` 是 `out` 对象的一个方法，其作用是在系统标准输出上显示形参里指定格式的字符串，并回车换行。

2. 对已创建好的Java源程序（.java）进行编译

用 Java 编译器对 Java 源程序（.java）进行编译，生成对应的字节码程序（.class）。如果编译成功，会得到一个有相同文件名的带.class 扩展名的字节码文件。

其命令格式如下：

```
javac HelloWorldApp.java
```

如果编译中不出现错误，将会得到一个名为 `HelloWorldApp.class` 文件。编译选项使用的是默认方式。

3. 解释执行已编译成功的字节码程序（.class）

用 Java 解释器 `java` 对 Java 字节码程序（.class）解释执行。

对于上例得到的 `HelloWorldApp.class` 文件，现在可以用 `java` 解释执行了，其命令格式如下：

```
java HelloWorldApp
```

运行的结果，将会在标准输出设备上输出：

```
Hello, Java World !
```

Java 解释器在解释执行时，解释处理的是类名，而不是文件名，所以在解释器 `java` 后面跟随的是类名，而不能写成文件名的形式（`HelloWorldApp.class`）。其选项也使用的默认方式。

以上简单介绍了不依赖浏览器而独立运行的 Java Application 应用程序，从编写到运行的基本步骤。如果想了解更详细的内容，请参阅后面相关章节的内容。

11.5.2 一个简单的Java Applet程序

Java Applet 程序是在 Java 兼容浏览器上执行的，它的创建和使用操作步骤与 Application 的不同。

1. 创建保存HTML页的目录

创建的这个目录是用来保存 HTML 页的，可以取名为 `HTML`。运行的平台不同，命令格式不同。

在 UNIX 系统下：

```
mkdir -/HTML  
cd -/HTML
```

在 Windows 95/NT 的 DOS 方式下：

```
md \HTML  
cd \HTML
```



如果已经有了存放 HTML 页的目录，就不需要这一步骤了。

2. 创建Java的Applet源程序（.java文件）

用字符编辑器输入相应的 Applet 源程序，它的程序中并不需要一个 `main()` 方法，但必须有一个类是系统类 `Applet` 的子类，就是说，必须有一个类的类头部分以 `extends Applet` 结尾。其中，`Applet` 是父类名，它是一个系统类；`extends` 是关键字，代表新定义的类是父类的子类。

系统类 `Applet` 中已经定义了很多域和方法，它们规定了 `Applet` 如何与执行它的解释器——Web 浏览器交互工作。当用户使用 `Applet` 的子类时，因为继承性，可以拥有父类的域和方法，从而使浏览器可以顺利执行用户程序定义的功能。

下面的例子是创建并保存在 HTML 目录中的一个文件名为 `HelloJavaApp.java` 的 Applet 小应用程序。

```
import java.applet.Applet;
import java.awt.Graphics;
public class HelloJavaApp extends Applet
{
    public void paint(Graphics g){
        g.drawString("Hello, Java Applet World !",50,25);
    }
}
```

注意：在上面的源程序中使用了 `import` 语句，这是因为我们要使用系统提供的 `Applet` 类和 `Graphics` 类，这两个类分别属于 `java.applet` 包和 `java.awt` 包。包（package）是类和接口的集合，在后面的章节中会详细介绍包的概念。

上例中，只对父类的 `paint` 方法进行了重载，该方法里面只有一条语句：

```
g.drawString("Hello, Java Applet World !",50,25);
```

`paint` 方法里有一个形式参数 `g`，是系统类 `Graphics` 的对象，`drawString` 是 `g` 对象的方法。它代表了在 Web 页面的 Applet 程序的界面区域上，调用 `g` 的 `drawString` 方法在（50,25）的地方显示字符串“Hello, Java Applet World !”。

3. 对已创建好的Applet源程序（.java）进行编译

用 Java 编译器对 Java 的 Applet 源程序（.java）进行编译生成对应的字节码程序（.class）。

如果编译成功，会得到一个有相同文件名的带 .class 扩展名的字节码文件。这与 Application 文件的编译是一样的。

其命令格式如下：

```
javac HelloJavaApp.java
```

得到一个名为 `HelloJavaApp.class` 的文件。编译选项使用的是默认方式。

4. 创建Applet，嵌入到HTML文档中

把编好的 Applet 小应用程序，嵌入到 HTML 文档中，并保存在一个文件中。下面创建一个名为 HelloWorld.HTML 的文件，并且将 HelloJavaApp.class 嵌入进去。

```
<HTML>
<HEAD>
<TITLE> A Simple Program </TITLE>
</HEAD>
<BODY>
<APPLET CODE="HelloJavaApp.class" WIDTH=500 HEIGHT=200>
</APPLET>
</BODY>
</HTML>
```

5. 加载HTML文件

在 Java 兼容浏览器的 URL 栏中，按照下面格式输入新建的 HTML 文件：

C:\HTML\HelloWorld.HTML

运行结果如图 11-8 所示。

另外，JDK 软件包中提供了一个模拟 Web 浏览器运行 Applet 应用程序的工具 AppletViewer，使用它调式程序，就不必反复调用庞大的浏览器了，其运行方式如下：

AppletViewer HelloJava.html

运行结果如图 11-9 所示。

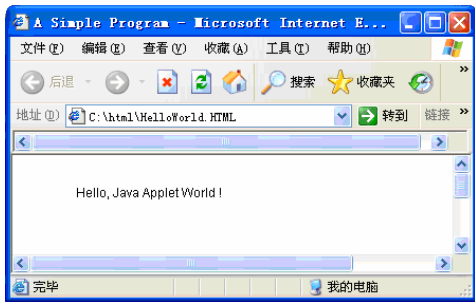


图 11-8 浏览器运行结果

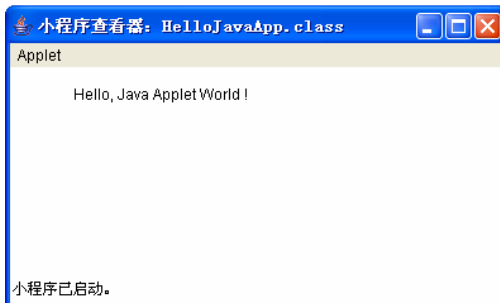


图 11-9 AppletViwer 运行结果

11.5.3 Java Applet图形界面的输入/输出

Java Applet 程序需要在浏览器中运行，而浏览器本身就是图形用户界面的环境，所以 Java Applet 程序只能在图形用户界面下工作。下面的例子是一个文件名为 AppletInOut.java 的 Applet 小应用程序。

【例 11-1】AppletInOut.java 文件。

```
import java.applet.*;
import java.awt.*;
```



```
import java.awt.event.*;

public class AppletInOut extends Applet implements ActionListener
{
    Label prompt;
    TextField input,output;
    public void init()
    {
        prompt=new Label("请输入您的名字:");//定义标签
        input=new TextField(6);//定义文本框
        output=new TextField(20);//定义文本框
        add(prompt);
        add(input);
        add(output);
        input.addActionListener(this); //向事件监听者注册
    }
    public void actionPerformed(ActionEvent e)
    {
        output.setText(input.getText()+"欢迎进入 Java 世界!");
    }
}
```

上例的功能是在第一个文本框中接收用户输入的字符串。当用户输入完毕按回车后，程序捕获这个字符串并在后面加入“欢迎进入 Java 世界！”，组合成一个新的字符串，然后在另一个文本框中显示出来，运行结果如图 11-10 所示。

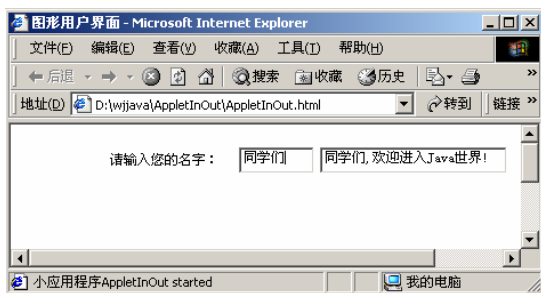


图 11-10 Java Applet 图形界面的输入/输出

程序的前 3 行分别加载了 Java 类库的三个包：`java.applet.*`、`java.awt.*`和 `java.awt.event.*`。凡是 Java.Applet 程序，必须加载 `java.applet.*`包；要使用图形用户界面，必须加载 `java.awt.*`包；要使用图形用户界面的事件处理，必须加载 `java.awt.event.*`包。

程序第 4 行定义了程序中唯一的类 `AppletInOut`。根据 Java.Applet 程序的规定，这个类应该是 `Applet` 类的子类，用 `extends Applet` 来说明。为了能够处理图形事件，用 `implements ActionListener` 来实现 `ActionListener` 接口，使之可以监听 `ActionEvent` 事件。

类 `AppletInOut` 里面的第 1、2 行定义了一个标签 (`Label`) 对象 `prompt` 和两个文本框 (`TextField`) 对象 `input` 与 `output`。`prompt` 用于输入提示信息, `input` 用于接收用户输入的信息, `output` 对象用于输出程序处理的结果信息。

类 `AppletInOut` 里面的第 3 行重载了 `init()` 方法, `init()` 方法在浏览器调用 `Java.Applet` 程序时自动执行。`init()` 方法里的第 1~6 行是创建 `prompt`、`input` 和 `output` 对象的实例, 并把它们添加到 `Applet` 程序的图形用户界面上。`init()` 方法里的最后一行是把 `input` 对象注册给 `Action` 事件的监听者, 使程序能响应用户在 `input` 中按回车键的操作。

类 `AppletInOut` 里面定义的另一个方法 `actionPerformed(ActionEvent e)`, 是让动作事件的监听者使用这个方法来处理动作事件的。因为在本例中只有 `input` 对象产生事件, 所以不用判断动作事件的来源, 直接用 `input.getText()` 方法在 `input` 文本框中获取用户输入的字符串, 用 “+” 与字符串 “, 欢迎进入 Java 世界!” 拼接在一起, 并利用 `output` 的 `setText` 方法把拼接后的字符串在 `output` 文本框中显示出来。

通过上例可以看出, 图形用户界面最基本的输入/输出手段是使用标签对象或文本框对象输出数据, 使用文本框对象输入数据。

11.5.4 Java Application 图形界面的输入/输出

本节编写一个图形用户界面的 `Java Application` 程序, 首先创建一个名为 `AppInOut.java` 的文件, 其源码如下。

【例 11-2】`AppInOut.java` 文件。

```
import java.awt.*;
import java.awt.event.*;
public class AppInOut
{
    public static void main(String args[])
    {
        new FrameInOut();
    }
}
class FrameInOut extends Frame implements ActionListener
{
    Label prompt;
    TextField input,output;
    Button btn;
    FrameInOut() {
        super("图形用户界面的 Java Application 程序");
        prompt=new Label("请输入您的名字: ");//定义标签
        input=new TextField(6); //定义文本框
        output=new TextField(20); //定义文本框
```



```
        btn=new Button("关闭");//定义按钮
        setLayout(new FlowLayout());//界面上的图形对象的布局策略
        add(prompt);
        add(input);
        add(output);
        add(btn);
        input.addActionListener(this); //向事件监听者注册
        btn.addActionListener(this); //向事件监听者注册
        setSize(300,200);
        show();
    }
    public void actionPerformed(ActionEvent e)
    {
        if(e.getSource()==input) //判断事件源
            output.setText(input.getText()+"欢迎进入 Java 世界!");
        else{
            dispose();
            System.exit(0);
        }
    }
}
```

然后对 AppInOut.java 文件进行编译：

```
javac AppInOut.java
```

如果编译中不出现错误，将会得到 AppInOut.class 和 FrameInOut.class 两个字节码文件。运行包含 main()方法的 AppInOut.class 文件，其运行结果如图 11-11 所示。

在上面的 AppInOut.java 程序中定义了两个类：FrameInOut 类是 java.awt 包中的窗框类 Frame 的子类，用于建立和使用图形用户界面；AppInOut 类是主类，它的 main 方法中创建了一个 FrameInOut 类的对象，即创建了一个图形界面的窗框。本例的功能与例 11-1 中的 Apple 程序功能基本相同，只是增加了一个“关闭”按钮。

与例 11-1 中的 init()方法类似，FrameInOut 类中的构造方法 FrameInOut()也在创建 FrameInOut 对象时自动调用执行，创建了一个标签、两个文本框和一个按钮。

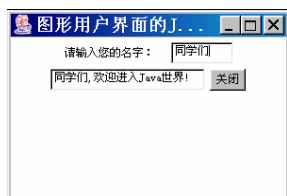


图 11-11 Java Application 图形界面的输入/输出

由于本例中增加了一个按钮 (Button) 对象 btn，因此除了文本框 input 对象外，按钮对象 btn 也要引发动作事件。所以在 actionPerformed 方法中，要判断动作事件的来源：如果是 input 产生的事件，就获取字符串并在 output 中显示；如果是 btn 产生的事件，就关闭窗口退出程序。

通过上面示例程序，介绍 Java 的 Application 和 Applet 程序设计的基本方法，为后面章节的学习打下了基础。



11.6 Java 符号集

符号是构成程序的基本单位,不同的语言所采用的符号标准有所不同,在 C 语言和 C++ 语言等语言中,一般都采用 ASCII 码;而 Java 则采用 Unicode 字符集,又称统一码字符集,它可以支持多种语言。

1. 关键字

关键字是构成编程语言本身的符号,是一种特殊的标识符,又称保留字。Java 语言中关键字有 40 多个,它们是:

abstract	boolean	break	byte	case	cast	catch	char
class	const	continue	default	do	double	else	extends
final	finally	float	for	future	generic	goto	if
implements	import	inner	instanceof	int	interface	long	native
new	null	operator	outer	package	private	protected	public
rest	return	short	static	super	switch	synchronized	this
throw	throws	transient	try	var	void	volatile	while

关键字有其特殊的意义,不能作为一般的标识符使用,即一般的标识符(变量名、类名、方法名等)不能与其同名。

2. 标识符

标识符是变量、类、方法等在程序中的唯一名字。标识符需要按照一定的规则取名。在 Java 语言中,标识符取名的规则如下:

- ◎ 必须由字母、下划线_或美元符 \$ 开头;
- ◎ 并由字母、0~9 的数字、下划线_或美元符 \$ 组成的;
- ◎ 不能与关键字名或布尔值(true 和 false)同名。

在 Unicode 字符集中,字母包括 26 个英文字母,还包括所有十六进制数值大于 00C0 的 Unicode 码的字符。允许标识符中使用来自不同字符集的字母。例如,希腊语、俄语、希伯来语等。

取名除了符合上述规则外,还要注意:

- ◎ 标识符名是具有一定的实际含义的一串字符,以便增强程序的可读性;
- ◎ 尽量少用除英文字母、下划线、美元符以外的字母,以减少录入难度;
- ◎ 少用美元符,以便于链接 C 代码时的处理。

标识符开头的字母以及标识符中间单词的第一个字母大写,而其余的字母都小写,最好不要用全部大写的标识符。一般,类名、接口名等的第一个字母大写,变量名、方法名等的第一个字母小写。这样与 Java 系统的使用习惯保持一致,例如, HelloWorldApp。

3. 程序中的注释

在程序中添加合理的注释,将大大增强可读性,为程序的调试和修改提供有益的帮助。



注释内容本身不对程序执行产生任何影响，只是文字注解。

Java 语言允许用 3 种形式在程序中注释。前两种与 C++ 语言的形式相同，而第 3 种是 Java 新增加的形式，如表 11-1 所示。

表 11-1 Java 语言的注释形式

形 式	作 用
//注释内容	由//开始到行末的内容均被系统忽略
/*注释内容*/	所有处在/*到*/之间的内容均被系统忽略
/**注释内容*/	在声明之前的所有处在/**到*/之间的内容均被系统忽略

另外，在 Java 语言中，有一个用于提取注释的工具：javadoc，对于采用/**注释内容*/形式的注释，可生成十分专业的程序文档。



11.7 数据的简单类型

在 Java 语言中，数据类型分为两大类：简单类型和引用（Reference）类型。

- ◎ 简单类型包括：整型、浮点型、字符型和布尔型等，具体如表 11-2 所示。
- ◎ 引用类型包括数组、类和接口，它是建立在其他类型之上的数据类型，是指向变量所代表的实际值或值的集合的引用。

表 11-2 简单数据类型

数 据 类 型	关 键 字	大 小(bit)	默 认 值
布尔型	boolean	8	false
字节型整数	byte	8	0
字符型	char	16	'\u0000'空格
短整型	short	16	0
整型	int	32	0
长整型	long	64	0
浮点型	float	32	0.0F
双精度型	double	64	0.0D

每一种基本数据类型都对应一种类，例如，double 类型对应 Double 类，char 类型对应 Character 类。这些类在包含基本数据类型所表示的一定范围、一定格式的数值的同时，还包含了一些特定的方法，可以实现对数值的专门操作。从这个意义上说，Java 仍继承了面向过程的一些东西。



11.8 常量

常量是指直接用于程序中的固定不变的值。

1. 布尔型常量

布尔型常量包括 `true` 和 `false`，表示“真”和“假”两种状态。

2. 整型常量

Java 整数类型常量有 3 种形式：十进制数、八进制数、十六进制数。

十进制整数由非 0 开头的数值表示，如 `100`，`-50`；

八进制整数由以 0 开头的数值表示，如 `017` 表示十进制数 15。

十六进制整数由以 `0x` 或 `0X` 开头的数值表示，如 `0X2F` 表示十进制数 47。

整型常量又可分为：一般整型常量（占 32 位）和长整型常量（占 64 位），长整型常量尾部有一个大写的 `L` 或小写的 `l`，如 `-286L`，`03356l`。

3. 浮点数

Java 浮点数是带有分数的十进制数，它又分为以 32 位形式存储的单精度数和以 64 位形式存储的双精度数两种，可以分别用 `f/F` 和 `d/D` 作为后缀来表示它们的类型。浮点数可以用小数点和科学记数法两种形式表示。例如，`3.14159265f`，`6.07e8F`，`9.08418e30d`，`9.7823e10D`。

4. 字符型常量

字符型常量用一对单引号括起来的单个 Unicode 字符表示。

可以用带“\”的字符来表示其中的一些不可显示或有特殊意义的字符，这类字符称为转义符。表 11-3 中列出的是 Java 语言中的转义符。

表 11-3 转义符

引用方法	对应 Unicode 码	含 义
<code>'\b'</code>	<code>'\u0008'</code>	退格
<code>'\t'</code>	<code>'\u0009'</code>	水平制表符 Tab
<code>'\n'</code>	<code>'\u000a'</code>	换行
<code>'\f'</code>	<code>'\u000c'</code>	表格符
<code>'\r'</code>	<code>'\u000d'</code>	回车
<code>'\"'</code>	<code>'\u0022'</code>	双引号
<code>'\''</code>	<code>'\u0027'</code>	单引号
<code>'\\'</code>	<code>'\u005c'</code>	反斜线

5. 字符串常量

字符串常量是用双引号括起来的一串字符，可以是 0 个，例如，“`Hello`”、“`My\nJava`”。

在 Java 中，可使用连接操作符“+”把两个或多个字符串常量串接起来，例如：

`“How are you?” + “\nI am fine.”`



11.9 变量

变量是程序运行中可变的数据，通常用来记录运算中间结果或保存数据。



1. 变量的声明

变量声明包括两个部分：变量的数据类型和变量的名称。声明语句如下：

```
type identifier[identifier];
```

其中，**type** 表示数据类型的关键字，它决定了变量能存储值的类型，以及对变量能进行何种操作；**identifier** 表示标识符，即变量名字。在变量声明语句中，可以同时为多个变量声明成相同的类型，它们之间用逗号分隔。

下面给出几个变量声明类型的例子：

```
char    mychar = 'W';  
long    mylong = -39884;  
int     count = 5643;  
double salary, tax
```

2. 变量的作用域

变量的作用域是指能够访问变量的代码块。变量一经被声明，它在被声明的程序块中有效，也就建立了变量的作用域。依照作用域，变量可分为 4 类。

- ◎ 成员变量：是一个类或对象的成员，它是类中声明，但不是类的方法中声明。
- ◎ 局部变量：在方法中或方法的一个代码块中声明。一般而言，局部变量自声明处开始，到本段代码块结束处均是有效的。
- ◎ 方法参数：方法参数以及构造方法参数，是用来为方法和构造方法传值的。
- ◎ 异常处理方法参数：是用来为异常处理方法传递值的。



11.10 运算符与表达式

表达式是指由变量、常量、对象、运算符和方法调用的，按照一定的运算规则组成的序列。



11.10.1 赋值运算与类型转换

1. 赋值运算符

赋值运算符就是把右操作数的值赋给左操作数。左操作数必须是一个变量，而右操作数是表达式：

变量或对象 = 表达式

2. 数据类型转换

如果赋值运算符两侧的操作数类型一致，就直接将右侧的数据赋给变量；如果不一致，就需要转换右侧的数据，然后再赋值给左侧的变量。

类型转换可分为自动类型转换和强制类型转换两种方式：

- ◎ 自动转换是指将数据自动地转换成目标类型格式的数据；
- ◎ 强制转换是指将数据显式地转换成目标类型格式的数据。

Java 的类型转换有严格的规定:将变量从占内存较小的短数据类型转化成占内存较多的长数据类型时,可自动转换;反之,则必须进行强制转换,其格式如下:

(数据类型) 变量名或表达式

例如:

```
byte MyByte = 10;
int MyInteger = -1;
MyInteger = MyByte           //自动转换
MyByte = (byte)MyInteger     //强制类型转换
//整型和浮点型可以相互转换
int x = 7;
float y;
y = (float)x/2;
```

11.10.2 算术运算符

Java 运算符又称操作符,它是对操作数进行运算的特定符号。每个运算符都有固定的操作数的数目。运算符按数目不同,可分为:

- ◎ 单目(一元)运算符,需要一个操作数;
- ◎ 双目(二元)运算符,需要两个操作数;
- ◎ 三目(三元)运算符,需要 3 个操作数。

1. 双目运算符

双目运算符和三目运算符,只能中缀使用。所谓中缀,是指运算符在操作数中间。常用的双目算术运算符如表 11-4 所示。

表 11-4 双目运算符

运 算 符	运 算	例 子
+	加	a+b
-	减	a-b
*	乘	a*b
/	除	a/b
%	取余	a%b

注意:

- ◎ 对整数类型的数据进行取余运算,比较有实际意义。
- ◎ 两个整数类型的数据相除时,结果是商数的整数部分,小数部分截去不要。若希望保留小数部分,则需要对两操作数进行强制类型转换。
- ◎ 浮点数的取余运算: $a \% b$ 相当于 $a - ((\text{int})(a/b) * b)$ 。



2. 单目运算符

单目运算符，有些既可前缀使用又可后缀使用，有些只能前缀使用。所谓前缀，是指运算符在操作数前面；而后缀是指运算符在操作数后面。常用的单目算术运算符如表 11-5 所示。

表 11-5 单目运算符

运 算 符	运 算	例 子	功 能 等 价
++	自加	a++或++a	a = a+1
--	自减	a--或--a	a = a-1
-	求相反数	-a	a = -a

【例 11-3】ArithOper.java 文件。

```
public class ArithOper
{
    public static void main(String args[])
    {
        int a=5+4;           //a=9
        int b=a*2;           //b=18
        int c=b/4;           //c=4
        int d=b-c;           //d=14
        int e=-d;            //e=-14
        int f=e%4;           //f=-2
        int g=3;
        int h=g++;           //h=3, g=4
        int i=++g;           //g=5, i=5
        double j=18.4;
        double k=j%4;        //k=2.4
        //a%b 相当于 a-(int(a/b) * b), 浮点型可以取余
        System.out.println(" a = " + a);
        System.out.println(" b = " + b);
        System.out.println(" c = " + c);
        System.out.println(" d = " + d);
        System.out.println(" e = " + e);
        System.out.println(" f = " + f);
        System.out.println(" g = " + g);
        System.out.println(" h = " + h);
        System.out.println(" i = " + i);
        System.out.println(" j = " + j);
        System.out.println(" k = " + k);
    }
}
```



```
    }  
}
```

运行程序：

```
java ArithOper
```

其运行结果为：

```
a = 9  
b = 18  
c = 4  
d = 14  
e = -14  
f = -2  
g = 5  
h = 3  
i = 5  
j = 18.4  
k = 2.3999999999999986
```

11.10.3 关系与逻辑运算

1. 关系运算符

关系运算符用于比较两个数据之间的大小关系，其结果是布尔值。假设 `opD1` 和 `opD2` 是两个可以用于关系运算符操作的操作数，表 11-6 中列出了 Java 关系运算符的使用方法。在 Java 语言中，任何类型的数据，无论简单类型数据还是引用类型数据，都可以使用 `==` 或 `!=` 运算符来比较两者是否相等或相同，这与 C 或 C++ 语言的关系运算符不一样。

表 11-6 关系运算符

运 算 符	功 能	说 明
>	大于	若 <code>opD1>opD2</code> 成立，则为真；否则为假
>=	大于等于	若 <code>opD1>=opD2</code> 成立，则为真；否则为假
<	小于	若 <code>opD1<opD2</code> 成立，则为真；否则为假
<=	小于等于	若 <code>opD1<=opD2</code> 成立，则为真；否则为假
==	相等	若 <code>opD1==opD2</code> 成立，则为真；否则为假
!=	不相等	若 <code>opD1!=opD2</code> 成立，则为真；否则为假

例如：

```
int x = 5, y = 7;  
boolean b = (x==y);
```



2. 逻辑运算符

Java 逻辑运算符是针对布尔型数据进行处理的操作符，其结果仍旧为布尔型数据，包括有逻辑的与、或、非操作，使用方式如表 11-7 所示。

表 11-7 逻辑运算符

运 算 符	使 用 方 法	功 能	说 明
&	opB1 & opB2	与	运算式若 opD1, opD2 全真则真，有假则假
&&	opB1 && opB2	简捷与	运算式若 opD1, opD2 全真则真，有假则假
	opB1 opB2	或	运算式若 opD1, opD2 有真则真，全假则假
	opB1 opB2	简捷或	运算式若 opD1, opD2 有真则真，全假则假
!	! opB1	非	运算式若 opD1 是真则为假，是假则为真

“&&”和“||”被称为简捷与和简捷非，是因为运算符右边的表达式可能被忽略不加以计算，而“&”和“|”两边的表达式都会被执行。

例如：

```
int x = 3, y = 5;
```

```
Boolean b = x > y && x++ == y--;
```

在计算 b 的取值时，先计算&&左边的关系表达式 $x > y$ ，其结果为假，根据逻辑与运算规则，只有表达式两边值都为真时，最后结果才为真；所以不论&&右边表达式结果如何，整个表达式的值都为假，右边的表达式就不予计算执行了；最后变量的取值为： $x = 3, y = 5, b = \text{false}$ 。

如果把上例中的简捷与（&&）换成与（&），最后变量的取值为： $x = 4, y = 4, b = \text{false}$ 。



11.10.4 位运算

位运算符对操作数以二进制比特位为单位进行操作和运算，操作数和结果都是整型数，其使用方法如表 11-8 所示。

表 11-8 位运算符

运 算 符	使 用 方 法	说 明
>>	opBt1 >> opBt2	opBt1 右移 opBt2 位
<<	opBt1 << opBt2	opBt1 左移 opBt2 位
>>>	opBt1 >>> opBt2	opBt1 无符号右移 opBt2 位
&	opBt1 & opBt2	opBt1 和 opBt2 按位与
	opBt1 opBt2	opBt1 和 opBt2 按位或
^	opBt1 ^ opBt2	opBt1 和 opBt2 按位异或
~	~ opBt1	opBt1 按位取反

注意：如果操作的对象是 char、byte、short，在位移动作发生前其值会自动晋升为 int，运算结果也为 int。

11.10.5 其他运算符

1. 三目条件运算符 (?:)

它的一般形式为：

`ExprBool ? Expression1 : Expression2`

其中，`ExprBool` 为布尔运算式，它的值为 `true` 时，执行 `Expression1`，否则执行 `Expression2`。

例如：

```
int x=5, y=8, z=2;
int k = x<3? y : z;    //k=z=2
int j = x>0? x :-x     //y 为 x 的绝对值
```

2. 扩展赋值运算符

它是在“=”的前面加上其他的运算符而构成的，有如下形式：

`var op= expression`

相当于：`var = var op expression`

其中，`var` 表示变量，`op` 为运算符，`op=` 为扩展赋值运算符，`expression` 为运算式。

例如：

```
int a = 3;
int b = 6;
a += b; //相当于：a = a + b
a *= b; //相当于：a = a * b
```

3. 对象运算符

对象运算符 `instanceof`，用来测定一个对象是否属于某个特定类或其子类的实例，是，则返回 `true`，否则返回 `false`。例如：

```
Boolean b = MyObject instanceof
TextField
```

11.10.6 优先级

运算符的优先级决定了表达式中不同运算执行的先后顺序，表 11-9 中按从高到低的顺序描述了各种运算符的优先级。

表 11-9 运算符优先级

最高优先级	. [] () ;
单目运算	- ~ ! ++ -- 强制类型转换符
加减运算	+ -
移位运算	>> << >>>
乘除运算	* / %
大小关系	< <= > >=
相等关系	== !=
与	&
异或	^
简捷与	&&
简捷或	
或	
三目条件运算符	?:
赋值	=



11.11 流程控制语句

流程控制语句是用来控制程序中各语句执行顺序的语句，可以把单个语句组合成能完成



一定功能的小逻辑模块。

其流程控制方式采用结构化程序设计中规定的三种基本流程结构，即：顺序结构、分支结构和循环结构，如图 11-12 所示。

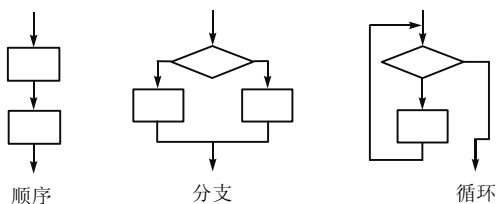


图 11-12 三种基本流程结构

11.11.1 分支语句

Java 中的分支语句有两个：一个是负责实现双分支的 if 语句，另一个是负责实现多分支的开关语句 switch。

1. if 语句

if 语句的一般形式是：

if (条件表达式)

语句块; //if 分支

else

语句块; //else 分支

其中，条件表达式用来判断程序的走向，如果表达式值为真，则执行 if 分支的语句块；否则执行 else 分支的语句块；也可以不书写 else 分支，若表达式为假，则绕过 if 分支直接执行 if 语句后面的其他语句。

【例 11-4】比较两个数的大小，并按从小到大的顺序输出。

```
public class CompareTwoDat
{
    public static void main(String args[])
    {
        double douDat1=4.18;
        double douDat2=9.8;
        if (douDat1 <= douDat2)
            System.out.println(douDat1+" <= "+douDat2+"\n");
        else
            System.out.println(douDat2+" >= "+douDat1+"\n");
    }
}
```

运行程序：

```
java CompareTwoDat
```

结果为：

```
4.18 <= 9.8
```

【例 11-5】根据考试成绩划分等级，90 分以上为 A，80~90 分为 B，等等。

```
class ScoreLevel
{
    public static void main(String args[])
    {
        int testScore=97;
        char grade;
        if (testScore >= 90)
            grade='A';
        else if (testScore >=80)
            grade='B';
        else if (testScore >= 70)
            grade='C';
        else if (testScore >= 60)
            grade='D';
        else
            grade='E';
        System.out.println(testScore+" is "+grade+"\n");
    }
}
```

运行程序：

```
java ScoreLevel
```

结果为：

```
97 is A
```

2. switch 语句

switch 语句的一般格式如下：

```
switch (表达式)
{
    case 判断值 1: 语句块 1           //分支 1
    case 判断值 2: 语句块 2           //分支 2
    .....
    case 判断值 n: 语句块 n           //分支 n
    default:      语句块 n+1         //分支 n+1
}
```



switch 语句在执行时，首先计算表达式的值，这个值必须是整型或字符型的，同时，应与各个 case 分支的判断值的类型相一致。计算出表达式值后，依次与各 case 语句进行比较，若相同，则执行相应分支语句；若都不相同，则执行 default 分支；若无 default 分支，则跳出整个 switch 语句。

【例 11-6】用 switch 和 break 语句，根据考试成绩划分等级。

```
class ScoreLevel
{
    public static void main(String args[])
    {
        int testScore=97,level;
        char grade=' ';
        level=testScore/10;
        switch (level)
        {
            case 10:
            case 9: grade='A';
                    break;
            case 8: grade='B';
                    break;
            case 7: grade='C';
                    break;
            case 6: grade='D';
                    break;
            case 5:
            case 4:
            case 3:
            case 2:
            case 1:
            case 0: grade='F';
        }
        System.out.println(testScore+" is "+ grade +"\n");
    }
}
```

运行程序：

```
java ScoreLevel
```

结果为：

```
97 is A
```

【例 11-7】请仔细阅读下面的 E.java 源文件，并分析运行结果。

```
public class E
{
    public static void main(String args[])
    {
        char c='\0';
        for (int i=1;i<=4;i++)
        {
            switch(i)
            {
                case 1:
                    c='b';
                    System.out.print(c);
                case 2:
                    c='e';
                    System.out.print(c);
                    break;
                case 3:
                    c='p';
                    System.out.print(c);
                default:
                    System.out.print("!");
            }
        }
    }
}
```

运行程序：

```
java ScoreLevel
```

结果为：

```
beep!!
```

11.11.2 循环语句

Java 的循环语句共有三种：**while** 语句、**do-while** 语句和 **for** 语句。它们的条件表达式的返回值都是布尔型的，循环体可以是单条语句，也可以是复合语句。

1. while 语句

while 语句的一般语法格式如下：

```
while (条件表达式)
```



循环体

`while` 语句先判断条件表达式的值，若为真，则执行循环体；若为假，则跳出循环，执行 `while` 语句后面的语句。

2. do-while语句

do 语句的一般语法格式如下：

do

循环体

while (条件表达式);

`do-while` 语句先执行一次循环体后，再判断条件表达式的值，若为真，则执行循环体；若为假，则跳出循环，执行 `do-while` 语句后面的语句。

3. for语句

for 语句一般语法格式如下：

for (表达式 1; 表达式 2; 表达式)

循环体

其中，表达式 1 用来完成变量初始化工作；表达式 2 为返回布尔值的条件表达式，若为真，则执行循环体；表达式 3 用来修改循环变量。

下面分别用 `while` 语句、`do-while` 语句、`for` 语句编程，输出以 5℃ 为间隔的摄氏温度到华氏温度的转换表。

【例 11-8】用 `while` 语句实现。

```
class TempConver_While
{
    public static void main(String args[])
    {
        int fahr,cels;
        System.out.println("Celsius    Fahrenheit\n");
        cels=0;                      //initialization
        while (cels<=30)             //condition
        {
            fahr=cels*9/5+32;        //body
            System.out.println(cels+"\t\t "+fahr);
            cels += 5;               //iteration
        }
    }
}
```

【例 11-9】用 `do-while` 语句实现。

```
class TempConver_DoWhile
{

```

```

public static void main(String args[])
{
    int fahr,cels;
    System.out.println("Celsius    Fahrenheit\n");
    cels=0;
    do
    {
        fahr=cels*9/5+32;
        System.out.println(cels+"\t\t"+fahr);
        cels += 5;
    } while (cels<=30);
}
}

```

【例 11-10】用 for 语句实现。

```

class TempConver_For
{
    public static void main(String args[])
    {
        int fahr,cels;
        System.out.println("Celsius    Fahrenheit\n");
        for (cels=0; cels<=30; cels += 5)
        {
            fahr=cels*9/5+32;
            System.out.println(cels+"\t\t"+fahr);
        }
    }
}

```

运行程序：

java TempConver_While (或 TempConver_DoWhile 或 TempConver_For)

结果为：

Celsius	Fahrenheit
0	32
5	33
10	34
15	35
20	36
25	37
30	38



11.11.3 跳转语句

Java 支持三种跳转语句：`continue` 语句、`break` 语句和 `return` 语句。

1. `continue` 语句

`continue` 语句必须用于循环结构中，它有两种使用形式。

一种是不带标号的 `continue` 语句，它的作用是终止当前这一轮的循环，跳过本轮剩余语句，直接进入当前循环的下一轮。

另一种是带标号的 `continue` 语句，其格式为：

`continue` 标号名

这个标号名应该定义在程序中外层循环语句的前面，用来标志这个循环结构。

【例 11-11】查找 1~100 之间的素数。

```
class Number
{
    public static void main(String args[])
    {
        First_Loop:
        for(int i=2; i<100; i++)
        {
            for(int j=2; j<i; j++)
            {
                if(i%j==0)
                    continue First_Loop;
            }
            //测试有没有比 i 小的数，可被除尽
            System.out.println(i); //打印素数
        }
        //end of main method
    }
    //end of class
}
```

2. `break` 语句

`break` 语句的作用是使程序的流程从一个语句块内部或循环体内部跳转出来。它也有两种使用形式。一种是不带标号的，用于 `switch` 语句。另一种是带标号的其格式是：

`break` 标号名

这个标号名用来标志某个语句块。执行 `break` 语句就从这个语句块中跳出来，流程进入其后面的语句。

3. `return` 语句

`return` 语句的一般格式是：

`return` 表达式

`return` 语句用来使程序流程从方法调用中返回，表达式的值就是调用方法的返回值。

习题 11

- 11.1 下载并安装 JDK 软件包，尝试阅读其中的 JDK 文档。
- 11.2 怎样区分 Java Application 和 Applet 程序。
- 11.3 Java 语言有哪些主要特点。
- 11.4 Java 语言包含哪三个版本，每个版本有什么功能，各个版本的应用领域？
- 11.5 编写一个 Java Application，利用 JDK 软件包中的工具编译并运行这个程序，要求在屏幕上显示 “Welcome to Java World!”。
- 11.6 编写一个 Java Applet，使之能在浏览器窗口中显示 “Welcome to Java applet World!”。
- 11.7 一个数如果恰好等于它的因子之和，这个数就称为“完数”。编写一个程序，输出 1000 之内的所有完数。

第 12 章 Java语言程序设计



12.1 Java的类和对象

类是 Java 语言的最基本概念，是组成 Java 程序的基本要素。类是 Java 的执行单位，Java 运行的就是 Java 类本身，它封装了该类对象的变量和方法。

对象是类的实例化，对象的创建是通过对象构造方法来实现的。我们可以生成多个对象，通过消息传递来进行交互，最终完成复杂的任务。消息传递是指激活指定的某个对象的方法，以改变它的状态或使其产生一定的动作。

类和对象的关系如图 12-1 所示，图中，左上角是现实世界中的一个实际的对象——小汽车；右边定义了计算机世界中对象的原型类——Car；左下角在一个代码段中使用了右边定义的类 Car，在计算机世界里生成了一个与现实世界对应的对象 myCar。

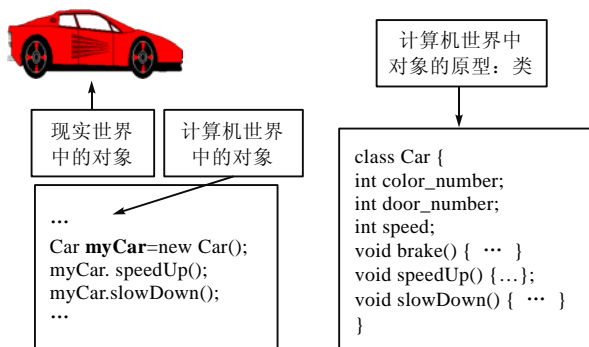


图 12-1 类和对象



12.1.1 系统定义的种类

Java 程序设计就是定义类的过程，它分为两大类：系统定义的种类，即 Java 类库中的类；用户程序自定义的类。要提高编程效率和质量，必须熟练掌握尽可能多的 Java 类库，根据功能的不同，Java 类库被划分成若干不同的包，每个包中都有不少具有特定功能和关系的类和接口。

下面列出一些常用的包。

- © java.lang 包。该包是 Java 语言的核心类库，包含了运行 Java 程序必不可少的系统类，如基本数据类型、基本数学函数、字符串处理、线程、异常处理类等。每个 Java 程序运行时，系统都会自动引入 java.lang 包，所以这个包的加载是默认的。
- © java.io 包。该包是 Java 语言的标准输入/输出类库，包含了实现 Java 程序与操作系统、

用户界面以及其他 Java 程序进行数据交换所使用的类，如基本输入/输出流、文件输入/输出流、过滤输入/输出流、管道输入/输出流、随机输入/输出流等。

- ◎ `java.util` 包。该包包含了 Java 语言中的一些低级的使用工具，如处理时间的 `Date` 类、处理变长数组的 `Vector` 类，实现栈和散列表的 `Stack` 类和 `HashTable` 类等。
- ◎ `java.awt` 包。该包是 Java 语言用来构建图形用户界面（GUI）的类库，它包含了许多界面元素和资源，主要提供三方面的支持：低级绘图操作，如 `Graphics` 类等；图形界面组件和布局管理，如 `Checkbox` 类、`Container` 类、`LayoutManager` 接口等；界面用户交互控制和事件响应，如 `Event` 类。
- ◎ `java.awt.image` 包。该包用来处理和操纵来自网上的图片的 Java 工具类库。
- ◎ `java.awt.peer` 包。该包的作用是使同一个 Java 程序在不同的软、硬件平台上运行时，具有相同的用户界面；它是程序代码和平台之间的中介，将不同的平台包裹、隐藏起来，使之在用户面前呈现相同的面貌；该包是实现 Java 语言跨平台特性的手段之一。
- ◎ `java.applet` 包。该包是用来实现运行于 Internet 浏览器中的 Java Applet 的工具类库，它仅包含少量的几个接口和一个非常有用的类：`java.applet.Applet`。
- ◎ `java.net` 包。该包是 Java 语言用来实现网络功能的类库，主要包括：底层的网络通信，如 `Socket` 类、`ServerSocket` 类；编写用户自己的 Telnet、FTP、邮件服务等实现网上服务的类；用于访问 Internet 上资源和进行 CGI 网关调用的类，如 `URL` 等。
- ◎ `java.corba` 包和 `java.corba.orb` 包。这两个包将 CORBA 嵌入到 Java 环境中，使 Java 程序可以存取、调用 CORBA 对象，并与之共同工作。
- ◎ `java.lang.reflect` 包。该包提供用于反射对象工具，允许程序监视一个正在运行的对象并获得它的构造函数、方法和属性。
- ◎ `java.rmi` 包、`java.rmi.registry` 包和 `java.rmi.server` 包。实现远程方法调用，可在远程计算机上创建对象，在本地计算机上使用该对象。
- ◎ `java.security` 包、`java.security.acl` 包和 `java.security.interfaces` 包。提供安全控制和管理，可对 Java 程序加密。
- ◎ `java.util.zip` 包。用来实现文件压缩功能的。
- ◎ `java.awt.datatransfer` 包。提供了处理数据传输的工具类，包括剪贴板、字符串发送器等。
- ◎ `java.sql` 包。是实现 JDBC（Java database connection）的类库。利用它可使 Java 程序具有访问不同种类数据库的能力，如 Oracle、Informix、Sybase、SQL Server、DB2 等。

12.1.2 用户程序自定义类

类的一般格式是：

```
classHead
{
    classBody
}
```



一个类的定义包含有两个部分的内容：`classHead` 类头的声明和 `classBody` 类体的定义。下面的程序段定义了一个电话卡类。

【例 12-1】

```
class PhoneCard
{
    long cardNumber;        //卡号
    private int password;    //密码
    double balance;         //剩余金额
    String connectNumber;    //接入号码（200 卡是 200，201 卡是 2011）
    boolean connected;

    boolean performConnection(long cn,int pw)
    {
        if(cn==cardNumber&&pw==password)
        {
            connected=true;
            return true;
        }
        else
        {
            connected=false;
            return false;
        }
    }
    double getBalance()
    {
        if(connected)
            return balance;
        else
            return -1;
    }
    void performDial()
    {
        if(connected)
            balance-=0.5;
    }
}
```

例 12-1 的程序片断定义了一个用户类 `PhoneCard`，第 1 句定义了类头，第 2 句到最后一

句定义了类体。

类头使用关键字 `class` 标志类定义的开始, `class` 后面跟着类名; 类体用一对大括号括起, 包括属性 (域) 和方法两大部分, 其中域对应类的静态属性, 方法对应类的行为和操作。

12.1.3 创建对象与定义构造函数

1. 创建对象

创建对象的一般格式为:

类名 新建对象名 = new 构造函数 ()

例如:

```
PhoneCard myCard = new PhoneCard();
```

创建对象与声明基本数据类型的变量类似, 赋值号右边的 `new` 是为新建对象开辟内存空间的运算符。与变量相比, 对象占用的内存空间要大得多, 对象是以类为模板创建的具体实例。

以 `PhoneCard` 类为例, 它定义了 5 个域和 3 个方法, 它的对象 `myCard` 的内存空间保存的域和方法分别为:

```
myCard.cardNumber, myCard.password, myCard.balance,
myCard.connectNumber, myCard.connected,
myCard.performConnection();
myCard.getBalance();
myCard.performDial();
```

要访问或调用一个对象的域或方法, 首先访问该对象, 然后用算符 “.” 连接该对象的域或方法。例如:

```
myCard.balance = 50;
```

2. 构造函数

构造函数是与类同名的方法, 创建对象的语句用 `new` 运算符开辟了新建对象的内存空间后, 将调用构造函数初始化这个新建对象。

构造函数是类的一种特殊方法, 其特殊性如下:

- ◎ 构造函数的方法名与类名相同;
- ◎ 构造函数没有返回值;
- ◎ 构造函数的作用是对类的对象进行初始化;
- ◎ 在创建一个类的新对象的同时, 系统会自动调用该类的构造函数。

例如, 定义 `PhoneCard` 类的构造函数如下:

```
PhoneCard(long cn, int pw, double b, String s)
{
    cardNumber = cn;
    password = pw;
```



```
        if(b>0)
            balance = b;
        else
            System.exit(1);
        connectNumber = s;
        connected = false;
    }
```

然后就可以用如下的语句初始化 PhoneCard 对象：

```
PhoneCard myCard = new PhoneCard(20010922, 342323, 50, "200")
```

【例 12-2】使用 PhoneCard 类的一个完整例子：UsePhoneCard.java。

```
public class UsePhoneCard
{
    public static void main(String args[])
    {
        PhoneCard myCard = new PhoneCard(280501,342323,50,"200");
        System.out.println(myCard.toString());
    }
}

class PhoneCard
{
    long cardNumber;
    private int password;
    double balance;
    String connectNumber;
    boolean connected;

    PhoneCard(long cn, int pw, double b, String s)
    {
        cardNumber = cn;
        password = pw;
        if(b>0)
            balance = b;
        else
            System.exit(1);
        connectNumber = s;
        connected = false;
    }

    boolean performConnection(long cn,int pw)
```

```

{
    if(cn==cardNumber&&pw==password)
    {
        connected=true;
        return true;
    }
    else
    {
        connected=false;
        return false;
    }
}
double getBalance()
{
    if(connected)
        return balance;
    else
        return -1;
}
void performDial()
{
    if(connected)
        balance-=0.5;
}
public String toString()
{
    String s = "电话卡接入号码:"+ connectNumber +"\n 电话卡卡号:"+ cardNumber+
        "\n 电话卡密码:"+ password +"\n 剩余金额:"+ balance;
    if(connected)
        return(s+"\n 电话已接通");
    else
        return(s+"\n 电话未接通");
}
}

```

运行结果:

电话卡接入号码:200

电话卡卡号:280501

电话卡密码:342323

剩余金额:50

电话未接通



12.1.4 类的修饰符

Java 程序定义类时，可在 `class` 之前增加若干类的修饰符来限定所定义类的特性。类的修饰符分为访问控制符和非访问控制符，有些修饰符可用来修饰类中的域或方法。本节讨论类的非访问控制符。

1. 抽象类

凡是用 `abstract` 修饰符修饰的类称为抽象类，抽象类就是没有具体对象的概念类。例如，“鸟”是一个抽象类，它可以派生出若干子类，如“鸽子”、“燕子”、“麻雀”、“天鹅”等，“鸟”仅仅作为一个抽象的概念存在，它代表所有鸟的共同属性，任何一只具体的鸟都是由“鸟”经过特殊化形成的某个子类的对象。

又如，电话卡有很多类型：磁卡、IC 卡、IP 卡、200 卡、校园 201 卡。不同种类的电话卡有各自的特点，例如，磁卡和 IC 卡没有卡号和密码，使用 200 卡每次通话要多扣 0.1 元的附加费等。同时它们也拥有一些共同的特性，例如，每张卡都有剩余的金额，都有通话的功能，因此，可以定义一种抽象的电话卡类。

```
abstract class PhoneCard
{
    double balance;
    void performDial()
    { ... }
}
```

其他具体的电话卡，可由抽象的电话卡类继承而来，再加入自身的特性，如 IC 卡类、IP 卡类、200 卡类等。

2. 最终类

如果一个类被 `final` 修饰，则说明该类不可以有子类了。我们可以把有继承关系的类看成一棵倒长的树，所有类的父类是树根，每个子类是一个分支，被声明为 `final` 的类是叶结点，且不能再有子类了。

被定义为 `final` 的类通常是一些有固定作用的、用来完成某种标准功能的类，如 Java 系统定义好的用来实现网络功能的 `InetAddress`、`Socket` 等类都是 `final` 类。



12.2 域和方法

12.2.1 域

域是类和对象的静态属性，它可以是基本数据类型的变量，也可以是其他类（系统类或用户自定义类）的对象，因此，定义域的操作就是说明变量或创建对象的操作。与类相似，域的修饰符也包括访问控制符和非访问控制符。本节讨论域的非访问控制符。

1. 静态域

用 `static` 修饰的域就是静态域。静态域最本质的特点是：它们是类的域，不属于任何一个类的具体对象，它不保存在某个对象的内存区域中，而是保存在类的内存区域的公共存储单元。换句话说，对该类的具体对象而言，静态域是一个公共的存储单元。任何一个类的对象访问它，得到的都是相同的值；任何一个类的对象修改它，都是对同一内存单元进行操作。

【例 12-3】TestStaticField.java 源程序中，定义了类。

```
public class TestStaticField
{
    public static void main(String args[])
    {
        PhoneCard200 my200_1 = new PhoneCard200();
        PhoneCard200 my200_2 = new PhoneCard200();
        My200_1.additoryFee = 0.1;
        System.out.println("第二张 200 卡的附加费"+ My200_2.additoryFee);
        System.out.println("200 卡类的附加费"+ PhoneCard200.additoryFee);
    }
}

class PhoneCard200
{
    static String connectNumber = "200";
    //所有的 200 电话卡的接入号码都是 200
    static double additoryFee;
    //所有的 200 电话卡的电话附加费相同
    long cardNumber;
    int password;
    boolean connected;
    double balance;
}
```

运行结果为：

第二张 200 卡的附加费：0.1

200 卡类的附加费：0.1

例 12-3 验证了静态域是类中每个对象共享的域。

2. 静态初始化器

静态初始化器是由 `static` 引导的一对大括号括起来的语句组。它的作用与类的构造函数相似，都用来完成初始化的工作，但两者也有根本性的不同。

◎ 构造函数对每个新创建的对象进行初始化，而静态初始化器对类本身进行初始化。



◎ 构造函数是在用 `new` 运算符产生新对象时由系统自动执行的，而静态初始化器则是在它所属的类加载入内存时由系统调用执行的。

◎ 不同于构造函数，静态初始化器不是方法，没有方法名、参数列表。

从例 12-4 中，我们可以看到静态初始化器和静态域同时应用在程序中产生的效果。

【例 12-4】 TestStatic.java 源程序。

```
public class TestStatic
{
    public static void main(String args[])
    {
        PhoneCard200 my200_1 = new PhoneCard200();
        PhoneCard200 my200_2 = new PhoneCard200();
        System.out.println("第一张 200 卡的卡号: "+my200_1.cardNumber);
        System.out.println("第二张 200 卡的卡号: "+my200_2.cardNumber);
    }
}

class PhoneCard200
{
    static long nextCardNumber;           //静态域
    static String connectNumber = "200"; //静态域
    static double additoryFee;
    long cardNumber;
    int password;
    boolean connected;
    double balance;

    static                               //静态初始化器
    {
        nextCardNumber = 280501001;
    }

    PhoneCard200()                       //构造函数
    {
        cardNumber = nextCardNumber++;
    }
}
```

运行结果为：

第一张 200 卡的卡号：280501001

第二张 200 卡的卡号：280501002

3. 最终域

程序中经常需要定义各种类型的常量，并为它们取一个类似于变量的标识符名字，这样

就可以在程序中用这个名字来引用常量。**final** 就是用来修饰常量的修饰符。一个类的域如果被声明为 **final**，则它的取值在程序的整个执行过程中都不会改变。

例如：

```
static final String connectNumber = "200";
```

4. 易失域

如果一个域被 **volatile** 修饰，则说明该域可能同时被几个线程所控制和修改，即该域不仅仅为当前程序所掌控，还有其他的程序操作来影响和改变该域。

通常，**volatile** 用来修饰接受外部输入的域，如：表示当前时间的域，可由系统的后台线程随时修改。

12.2.2 方法

方法是类的动态属性，标志了类所具有的功能和操作，用来把类和对象的数据封装在一起。Java 的方法与其他语言中的函数或过程类似，是一段完成某种功能的程序段。

方法由方法头和方法体组成，一般格式如下：

```
修饰符 1 修饰符 2...返回值类型 方法名（形式参数列表）throws[异常列表]
{
    方法体个语句;
}
```

其中，形式参数列表的格式为：

```
形式参数类型 1 形式参数名 1, 形式参数类型 2 形式参数名 2, ...
```

1. 抽象方法

修饰符 **abstract** 修饰的抽象方法是一种仅有方法头，而没有具体的方法体和操作实现的方法。例如，在抽象类 **PhoneCard** 中定义一个抽象方法 **performDial()**：

```
abstract void performDial();
```

使用抽象方法可使所有 **PhoneCard** 类的子类对外呈现一个统一的接口。至于方法体的具体实现，则留到当前类的不同子类的类定义中完成。

注意：所有的抽象方法，都必须存在于抽象类之中。若一个抽象类的子类不是抽象类，则它必须为父类中的所有抽象方法书写方法体。例 12-5 说明了抽象方法的用法。

【例 12-5】TestAbstract.java 源程序。

```
public class TestAbstract
{
    public static void main(String args[])
    {
        PhoneCard200 my200 = new PhoneCard200(50);
        IC_Card myIC = new IC_Card(50);
```



```
        System.out.println("200 卡可拨打:"+my200.TimeLeft()+"次电话");
        System.out.println("IC 卡可拨打:"+myIC.TimeLeft()+"次电话");
    }
}
//抽象类定义
abstract class PhoneCard
{
    double balance;
    abstract void performDial();    //抽象方法
    int TimeLeft()                //测试余款还可打几次电话
    {
        double current = balance;
        int times;
        for(times=0;balance>0;times++)
            performDial();
        balance = current;
        return times;
    }
}
//定义 200 卡子类
class PhoneCard200 extends PhoneCard
{
    static long nextCardNumber;
    static String connectNumber = "200";
    static double additoryFee;
    long cardNumber;
    int password;
    boolean connected;
    static
    {
        nextCardNumber = 280501001;
    }
    PhoneCard200(double b)
    {
        cardNumber = nextCardNumber++;
        balance = b;
    }
    void performDial()
```

```

        { balance-=0.5; }
    }
    //定义 IC 卡子类
    class IC_Card extends PhoneCard
    {
        IC_Card(double b)
        { balance =b; }
        void performDial()
        { balance-=0.4; }
    }

```

运行结果为：

200 卡可拨打：100 次电话

IC 卡可拨打：126 次电话

2. 静态方法

用 `static` 修饰的方法是属于整个类的方法，它的特点如下。

- ◎ 调用该方法时，应使用类名作为前缀，而不是某个具体的对象名。
- ◎ `Static` 方法是属于整个类的方法，它在内存中的代码将随着类的定义而分配和装载，不被任何对象专有；非 `static` 方法是属于某个对象的方法，在对象创建时，在内存中拥有自己的专用代码段。
- ◎ `static` 不能操作属于某个对象的成员变量，而只能处理属于整个类的成员变量，即 `static` 方法只能处理 `static` 域。

例如，在 `PhoneCard200` 中需要修改附加费，可以定义一个 `setAdditory()` 的静态方法如下：

```

static void setAdditory(double newAdd)
{
    if(newAdd>0)
        additoryFee = newAdd;
}

```

3. 最终方法

用 `final` 修饰的方法是功能和内部语句不能被更改的最终方法。

在面向对象程序设计中，子类可改写从父类那里继承来的某个方法，形成新的类方法，其与父类方法同名，解决的问题类似，但具体实现和功能不全一致的新的类方法，这个过程称为重载。如果类方法被 `final` 修饰了，则该类的子类不能再重新定义与此方法同名的自己的方法，只能使用从父类继承来的方法。这样，可以防止子类对父类的关键方法的错误的重定义，保证了程序的安全性和正确性。

注意：所有被 `private` 修饰的方法，以及包含在 `final` 类中的方法，都被默认为是 `final` 的。



4. 本地方法

`native` 修饰符一般用来声明用其他语言编写的方法。

由于 `native` 方法是以非 Java 字节码的二进制代码形式嵌入 Java 程序的，所以整个 Java 程序的跨平台性能将受到限制或破坏，因此使用这类方法时应特别谨慎。

5. 同步方法

`synchronized` 修饰符主要用于多线程共存的程序中的协调和同步。

如果 `synchronized` 修饰的是类的方法（`static` 方法），则在被调用前将对应当前类的对象加锁。

如果 `synchronized` 修饰的是一个对象的方法（非 `static` 方法），则在被调用前将当前对象加锁。



12.3 访问控制符

访问控制符是一组限定类、域或方法是否可以被程序里的其他部分访问和调用的修饰符。类的访问符只有 `public`，域和方法的访问控制符有三个：`public`、`private` 和 `protected`。

1. 公共访问控制符 `public`

（1）类

Java 的类是通过包来组织的，处于同一包中的类可以不需任何说明而方便地互相访问和引用，而对于处于不同包中的类，默认设置为它们互相之间不可见。但是，当一个类被声明为 `public` 时，它就具有了被其他包中类访问的可能性，只要这些包中的类在程序中使用 `import` 语句引入 `public` 类，就可访问它了。

一个类作为整体可见，并不能代表类中的域和方法也一定可见，除非它们也被声明为 `public`。

C++ 中没有包的概念，所以没有类的修饰符。

（2）方法

被定义为 `public` 的方法是这个类对外的接口部分，程序的其他部分通过调用 `public` 方法达到与当前类交换信息，甚至影响当前类的目的。

如果一个类希望作为公共工具供其他的类和程序使用，则应该把该类和方法定义为 `public`。每个 Java 程序的主类都必须是 `public` 类，这是为了让其他类和程序可以访问它。

（3）域

用 `public` 修饰的域被称为公共域，如果一个公共域属于一个公共类，则它可被所有其他类所引用。`public` 修饰符会造成安全性和数据封装性下降，所以一般应减少 `public` 域的使用。

2. 默认访问控制符

如果一个类没有访问控制符，默认为该类只能被同一个包中的类访问，这种访问特性又称包访问性。

同理，类内的域和方法如果没有访问控制符，则说明它们具有包访问性，可以被同一个

包中的其他类所访问和调用。

3. 私有访问控制符private

用 `private` 修饰的域或方法只能被该类自身所访问和修改，而不能被任何其他类（包括该类的子类）来获取和引用。`private` 修饰符提供了最高的保护级别。

例如，在 200 电话卡类 `PhoneCard200` 中，电话卡的密码 `password` 不能允许其他类或对象随意查看或修改，所以这个域可以声明为私有的：

```
private int password;
```

当其他类希望获取或修改私有成员时，需要借助于类的方法来实现，例如，可以在类 `PhoneCard200` 中定义方法 `getPassword()` 来获取密码，定义方法 `setPassword()` 来修改密码，从而把 `password` 完全包裹保护起来。同时，为保证只有具备一定权限才能查看或修改密码，可在 `getPassword()` 方法和 `setPassword()` 方法中做必要的安全性检查，满足了一定条件才能获得或修改变量 `password` 的数值，从而保证了私有数据的私有性。

4. 保护访问控制符protected

用 `protected` 修饰的域和方法可以被以下三类引用：

- ◎ 该类自身；
- ◎ 与它在同一个包中的其他类；
- ◎ 在其他类中的该类的子类。

使用 `protected` 修饰符的主要作用是允许包中的类和其他包中的子类来访问父类的特定属性。

修饰符使用需要注意以下问题：

- ◎ `abstract` 不能与 `final` 并列修饰同一个类。
- ◎ `abstract` 不能与 `private`、`static`、`final` 或 `native` 并列修饰同一个方法。
- ◎ `abstract` 类中不能有 `private` 的成员（包括域和方法）。
- ◎ `abstract` 方法必须在 `abstract` 类中。
- ◎ `static` 方法中不能处理非 `static` 属性。



12.4 继承

当一个类拥有另一个类的数据和操作时，就称这两个类之间具有继承关系，被继承的类称为父类或超类，继承的类称为子类。一个父类可以同时拥有多个子类，该父类实际上是所有子类的公共域和公共方法的集合，而子类是父类的特殊化，可对公共域和方法在功能、内涵方面加以扩展和延伸。

在面向对象的继承特性中，还有一个关于单继承和多继承的概念。

- ◎ 单继承是指任何类都只有一个父类。
- ◎ 多继承是指一个类可以有一个以上的父类，该类继承了所有父类的静态数据和操作。

Java 语言只支持单重继承，但支持接口，一个类可以实现多个接口。利用接口可以得到多继承的优点，又没有多继承混乱、复杂的问题。



12.4.1 派生子类

Java 语言中的继承是通过 `extends` 关键字来实现的（C++语言中通过 `public` 和 `private` 实现）。在定义类时，用 `extends` 关键字指明新定义类的父类，就在两个类之间建立了继承关系。新定义的类被称为子类，它可以从父类那里继承所有非 `private` 的属性和方法作为自己的成员。

例如：

```
class 子类名 extends 父类名
```

图 12-2 所示为电话卡的继承关系树，有圆角的矩形框为抽象类，直角的矩形框为非抽象类。例 12-6 实现了图 12-2 中的各电话卡类。

【例 12-6】电话卡类及其子类的实现。

```
import java.util.*;

abstract class PhoneCard
{
    double balance;
    abstract boolean performDial(); //抽象方法
    double getBalance()           //获取余额
    { return balance; }
}

//定义无卡号电话卡子类
abstract class None_Number_PhoneCard extends PhoneCard
{
    String phoneSetType;
    String getSetType()
    { return phoneSetType; }
}

//定义有卡号电话卡子类
abstract class Number_PhoneCard extends PhoneCard
{
    long cardNumber;
    int password;
    String connectNumber;
    boolean connected;
    boolean performConnection(long cn,int pw)
    {
        if(cn==cardNumber&&pw==password)
        {
```

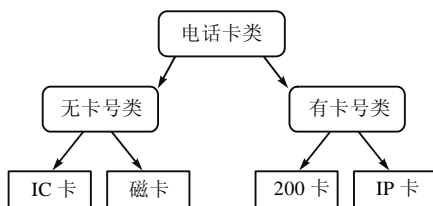


图 12-2 电话卡继承树


```
        connected=true;
        return true;
    }
    else
    {
        connected=false;
        return false;
    }
}
}
//定义磁卡子类
class MagCard extends None_Number_PhoneCard
{
    String usefulArea;
    boolean performDial()
    {
        if(balance>0.4)
        {
            balance-=0.4;
            return true;
        }
        else
            return false;
    }
}
//定义 IC 卡子类
class IC_Card extends None_Number_PhoneCard
{
    boolean performDial()
    {
        if(balance>0.4)
        {
            balance-=0.4;
            return true;
        }
        else
            return false;
    }
}
```



//定义 IP 卡子类

```
class IP_Card extends Number_PhoneCard
{
    Date expireDate;
    boolean performDial()
    {
        if(balance>0.3&&expireDate.after(new Date()))
            //new Date()返回当前日期
            //expireDate.after(new Date()),若失效日期比当前日期晚, 则返回 true, 否则返回 false
        {
            balance-=0.4;
            return true;
        }
        else
            return false;
    }
}
```

//定义 200 电话卡子类

```
class P200_Card extends Number_PhoneCard
{
    double additoryFee;
    boolean performDial()
    {
        if(balance>(0.3+additoryFee))
        {
            balance-=(0.3+additoryFee);
            return true;
        }
        else
            return false;
    }
}
```



12.4.2 域的继承与隐藏

1. 域的继承

子类可以继承父类的所有非私有域。下面以例 12-6 定义的电话卡类来说明域的继承。

PhoneCard 类:

```
double balance;
```

None_Number_PhoneCard 类:

```
double balance;           //继承自父类 PhoneCard
```

```
String PhoneSetType;
```

Number_PhoneCard 类:

```
double balance;           //继承自父类 PhoneCard
```

```
long cardNumber;
```

```
int password;
```

```
String connectNumber;
```

```
boolean connected;
```

MagCard 类:

```
double balance;           //继承自父类 None_Number_PhoneCard
```

```
String PhoneSetType;      //继承自父类 None_Number_PhoneCard
```

```
String usefulArea;
```

IC_Card 类:

```
double balance;           //继承自父类 None_Number_PhoneCard
```

```
String PhoneSetType;      //继承自父类 None_Number_PhoneCard
```

IP_Card 类:

```
double balance;           //继承自父类 Number_PhoneCard
```

```
long cardNumber;          //继承自父类 Number_PhoneCard
```

```
int password;             //继承自父类 Number_PhoneCard
```

```
String connectNumber;     //继承自父类 Number_PhoneCard
```

```
boolean connected;        //继承自父类 Number_PhoneCard
```

```
Date expireDate;
```

P200_Card 类:

```
double balance;           //继承自父类 Number_PhoneCard
```

```
long cardNumber;          //继承自父类 Number_PhoneCard
```

```
int password;             //继承自父类 Number_PhoneCard
```

```
String connectNumber;     //继承自父类 Number_PhoneCard
```

```
boolean connected;        //继承自父类 Number_PhoneCard
```

```
double additoryFee;
```

2. 域的隐藏

子类重新定义一个从父类那里继承来的域变量完全相同的变量，称为域的隐藏。

例如，把例 12-6 中对 200 电话卡子类的定义修改为：

```
//定义 200 电话卡子类
```

```
class P200_Card extends Number_PhoneCard
```

```
{
```



```
double additoryFee;
double balance; //父类已定义过
boolean performDial()
{
    if(balance>(0.3+additory))
    {
        balance-=(0.3+additory);
        return true;
    }
    else
        return false;
}
```

在上面的类定义中，增加定义了与从父类继承来的 `balance` 变量完全相同的变量，这样，`P200_Card` 类的域变为：

`P200_Card` 类：

```
double balance;           //继承自父类 Number_PhoneCard
double balance;           // P200_Card 类自己定义的域
long cardNumber;          //继承自父类 Number_PhoneCard
int password;             //继承自父类 Number_PhoneCard
String connectNumber;     //继承自父类 Number_PhoneCard
boolean connected;        //继承自父类 Number_PhoneCard
double additoryFee;
```

【例 12-7】域的隐藏，`TestHiddenField.java` 源文件。

```
public class TestHiddenField
{
    public static void main(String args[])
    {
        P200_Card my200 = new P200_Card();
        my200.balance = 50;
        System.out.println("父类被隐藏的金额为:"+my200.getBalance());
        //父类的方法操纵父类定义的域
        if(my200.performDial())
            System.out.println("子类的剩余金额为:"+my200.balance);
    }
}

abstract class PhoneCard
{
```

```

double balance;
abstract boolean performDial();    //抽象方法
double getBalance()                //获取余额
{
    return balance;
}
}
//定义有卡号电话卡子类
abstract class Number_PhoneCard extends PhoneCard
{
    long cardNumber;
    int password;
    String connectNumber;
    boolean connected;
    boolean performConnection(long cn,int pw)
    {
        if(cn==cardNumber&&pw==password)
        {
            connected=true;
            return true;
        }
        else
        {
            connected=false;
            return false;
        }
    }
}
//定义 200 电话卡子类
class P200_Card extends Number_PhoneCard
{
    double additoryFee;
    double balance;
    boolean performDial()
    {
        if(balance>(0.3+additoryFee))
        {
            balance-=(0.3+additoryFee);

```



```
        return true;
    }
    else
        return false;
    }
}
```

运行结果为：

父类被隐藏金额为：0.0

子类的剩余金额为：49.7



12.4.3 null、this与super

Java 系统默认每个类都具有三个域：null、this 与 super，所以在任意类中都可以不加说明而直接使用它们。

1. null

null 表示“空”变量，用于指代某一对象，但是，这个对象是不存在相应的实例的，例如：

```
Point pNull = null;
```

它创建的是 Point 类的变量 pNull，但不能创建相应的对象，即 pNull 是不属于任何实例对象的，相当是个“空壳”。

在方法需要用对象作为参数时，可以用 null 来代替。例如：

```
drawPoint(null);
```

2. this

this 表示的是当前对象本身，更准确地说，this 代表当前对象的一个引用。对象的引用可以理解为对象的另一个名字，通过引用可以顺利地访问到对象，包括访问、修改对象的域、调用对象的方法。在对象方法体中可直接引用对象的成员变量，然而，有时出现相同的名字，就需要明确成员变量名。

例如：

```
class HsbColor
{
    int hue,saturation,brightness;
    HsbColor (int hue, int saturation, int brightness)
    {
        this.hue = hue;           //色调
        this.saturation = saturation; //饱和度
        this.brightness = brightness; //明亮
    }
}
```

```
    }
}
```

在上例的构造方法中，用 **this** 来区分参数（不带 **this**）和成员变量（带 **this**）。当参数名与成员名相同时，Java 按参数优先并用相同的名字隐藏了成员变量，所以要指向成员变量，必须通过 **this** 明确当前对象中的成员。

当指向一个其方法作为引用出现的对象的成员变量时，使用 **this** 会增加代码的清晰度，减少基于名字相关的错误。也可以使用 **this** 调用当前对象的方法，这只有在方法名有歧义或需要增加代码清晰度时才是必要的。

3. super

子类对父类的同名成员变量和方法，分别实行隐藏和重载。但是，有时需要在子类中访问父类的变量，重载父类的方法。Java 语言提供的 **super** 就可以实现这访问。它有三种使用情况。

- ① 访问被隐藏的父类成员变量，如：**super.VariableName**。
- ② 调用父类中被重载的方法，如：**super.MethodName**。
- ③ 调用父类中的构造方法，如：**superClassName([paramList])**。

【例 12-8】Inheritance.java 源程序。

```
//父类
class SuperClass
{
    int x;
    SuperClass()    //构造函数
    {
        x=3;
        System.out.println("In SuperClass : x = "+x);
    }
    void doSomething()
    {
        System.out.println("In SuperClass.doSomething()");
    }
}

//子类
class SubClass extends SuperClass
{
    int x;
    SubClass()      //构造函数
    {
        super();    //父类构造函数
        x=5;
        System.out.println("In SubClass : x = "+x);
    }
}
```



```
    }  
    void doSomething()  
    {  
        super.doSomething();  
        System.out.println("In SubClass.doSomething()");  
        System.out.println("super.x="+super.x+"    sub.x="+x);  
    }  
}  
public class Inheritance  
{  
    public static void main(String args[])  
    {  
        subClass subC = new subClass();  
        subC.doSomething();  
    }  
}
```

运行结果为:

```
In superClass : x = 3  
In SubClass : x = 5  
In SuperClass.doSomething()  
In SubClass.doSomething()  
super.x = 3    sub.x = 5
```



12.5 多态性

多态性 (Polymorphism) 是面向对象程序设计的又一特殊性。面向过程的语言编程的主要工作是编写一系列过程或函数, 这些过程和函数各自对应一定的功能, 它们之间不能重名, 否则在调用时, 就会产生歧异和错误。而在面向对象的程序设计中, 有时却需要利用这样的“重名”现象来提高程序的抽象度和简捷性。



12.5.1 方法的继承

父类的非私有方法可以被子类继承。这里仍旧以例 12-5 定义的 PhoneCard 类及其子类为例来说明方法如何被继承。

PhoneCard 类:

```
abstract boolean performDial();  
double getBalance();
```

None_Number_PhoneCard 类:

```
abstract boolean performDial(); //继承自父类 PhoneCard
```



```

double getBalance();           //继承自父类 PhoneCard
Number_PhoneCard 类:
abstract boolean performDial(); //继承自父类 PhoneCard
double getBalance();           //继承自父类 PhoneCard
boolean performConnection(long cn, int pw)

MagCard 类:
double getBalance;             //继承自父类 None_Number_PhoneCard
String getSetType;             //继承自父类 None_Number_PhoneCard
boolean performDial()

IC_Card 类:
double getBalance;             //继承自父类 None_Number_PhoneCard
String getSetType;             //继承自父类 None_Number_PhoneCard
boolean performDial()

IP_Card 类:
double getBalance;             //继承自父类 Number_PhoneCard
boolean performConnection(long cn, int pw)
//继承自父类 Number_PhoneCard
boolean performDial()

P200_Card 类:
double getBalance;             //继承自父类 Number_PhoneCard
boolean performConnection(long cn, int pw)
//继承自父类 Number_PhoneCard
boolean performDial()

```

12.5.2 覆盖

子类可以定义与父类同名的方法，实现对父类方法的覆盖（Overload）。方法的覆盖与域的覆盖的区别为：子类隐藏父类的域只是使之不可见，父类的同名域在子类对象中仍然占有自己独立的内存空间；而子类方法对父类同名方法的覆盖将清除父类方法占用的内存，从而使父类方法在子类对象中不存在。

12.5.3 重载

方法的重载（Override）是实现多态技术的重要手段。与方法覆盖不同，重载不是子类对父类同名方法的重新定义，而是类对自身已有的同名方法的重新定义。由于重载发生在一个类里，不能用类名来区分不同的方法，所以采用不同的形式参数列表，包括形式参数的个数、类型、顺序的不同，来区分重载的方法。



【例 12-9】TestOverride.java 源程序。

```
public class TestOverride
{
    public static void main(String args[])
    {
        P200_Card my200 = new P200_Card(2004101601,135790,50);
        if(my200.performDial(2004101601,135790))
            System.out.println("拨打电话后剩余金额为:"+my200.getBalance());
        if(my200.performDial())
            System.out.println("拨打电话后剩余金额为:"+my200.balance);
    }
}

//定义父类
abstract class PhoneCard
{
    double balance;
    abstract boolean performDial(); //抽象方法
    double getBalance()           //获取余额
    {
        return balance;
    }
}

//定义有卡号电话卡子类
abstract class Number_PhoneCard extends PhoneCard
{
    long cardNumber;
    int password;
    String connectNumber;
    boolean connected;
    boolean performConnection(long cn,int pw)
    {
        if(cn==cardNumber&&pw==password)
        {
            connected=true;
            return true;
        }
        else
        {
            connected=false;
```

```

        return false;
    }
}
}
//定义 200 电话卡子类
class P200_Card extends Number_PhoneCard
{
    double additoryFee=0.1;
    P200_Card(long cn,int pw,double b)    //构造函数
    {
        cardNumber = cn;
        password = pw;
        balance = b;
    }
    boolean performDial()                //方法的重载
    {
        if(balance>(0.3+additoryFee))
        {
            balance-=(0.3+additoryFee);
            return true;
        }
        else
            return false;
    }
    boolean performDial(long cn,int pass) //方法的重载
    {
        if(performConnection(cn,pass))
            return performDial();
        else
            return false;
    }
    double getBalance()                  //对父类方法的覆盖
    {
        if(connected)
            return balance;
        else
            return -1;
    }
}

```



运行结果为：

拨打电话后剩余金额为：49.6

拨打电话后剩余金额为：49.2

12.5.4 构造函数的继承与重载

构造函数是类的一种特殊函数，它可以从父类继承，也可以互相重载。

1. 构造函数的继承

子类可以继承父类的构造函数，构造函数的继承遵循以下的原则：

- ◎ 子类无条件地继承父类的不含参数的构造函数。
- ◎ 若子类没有定义自己的构造函数，则继承父类无参数的构造函数作为自己的构造函数。
- ◎ 若子类定义了自己的构造函数，则先执行继承自父类的无参数构造函数，再执行自己的构造函数。
- ◎ 对父类含参数的构造函数，子类可以通过在定义自己的构造函数中使用 `super` 关键字来调用它，但这个调用语句必须是子类构造函数的第一个可执行语句。

2. 构造函数的重载

构造函数的重载是指同一个类中存在若干个具有不同参数列表的构造函数，创建该类对象的语句会自动根据给出的实际参数的数目、类型和顺序来确定调用哪个构造函数来完成对新对象的初始化工作。

一个类的若干个构造函数之间可以互相调用。当一个构造函数调用另一个时，可以使用 `this` 关键字，但该调用语句应是整个构造函数的第一个可执行语句。下面前 4 个构造函数是 `Number_PhoneCard` 类的 4 个重载的构造函数，最后一个是 `Number_PhoneCard` 类子类 `P200_Card` 的构造函数。

```
Number_PhoneCard()
{
}

Number_PhoneCard(long cn)
{
    cardNumber = cn;
}

Number_PhoneCard(long cn,int pw)
{
    cardNumber = cn;
    password = pw;
}

Number_PhoneCard(long cn,int pw,double b)
```

```

{
    this(cn,pw);
    balance = b
}

P200_Card(long cn,int pw,double b,double a)
{
    super(cn,pw,b);
    additoryFee = a;
}

```



12.6 上转型对象

假设 A 类是 B 类的父类，当我们用子类创建一个对象，而这个对象的引用放到父类的对象中时，有

```
A a;    a = new B();
```

或

```
A a;    B b = new B();    a = b;
```

称这个父类对象 a 是子类对象的上转型对象。例如，“老虎是哺乳动物”，哺乳类是老虎类的父类，但这样说将失掉老虎独有的属性。

上转型对象不能操作子类新增的域和方法。上转型对象可以操作子类继承或重写的域和方法。如果子类重写了父类的某个方法，则上转型对象调用该方法时，是调用的重写方法。

【例 12-10】ZhuanXing.java 源程序。

```

class ZhuanXing
{
    public static void main(String args[])
    {
        Anthropoid mon=new Monkey();    //mon 是上转型对象
        mon.crySpeak("I love apple.");
        Anthropoid ant=new Anthropoid();    //ant 是父类对象
        ant.crySpeak("I love apple.");
        Monkey mon1=new Monkey();    //mon1 是子类对象
        mon1.crySpeak("I love apple.");
        mon1.computer(8,10);
    }//end of main method
}//end of class

class Anthropoid
{

```



```
private int n=100;
void crySpeak(String s)
{
    System.out.println(s);
}
}
class Monkey extends Anthroid
{
    void computer(int a,int b)
    {
        int c=a*b;
        System.out.println(c);
    }
    void crySpeak(String s)
    {
        System.out.println("**"+s+"**");
    }
}
```

运行结果为：

```
**I love apple.**
I love apple.
**I love apple.**
80
```

注意：mon.computer(8,10);语句是不合法的，因为 computer 是子类新增的方法，上转型对象 mon 不能调用它。



12.7 接口

接口（Interface）也称为界面，在其声明语法上有些类似于类。实际上，完全可以把接口理解为一种特殊的类，一种由常量和抽象方法组成的特殊类。

在 Java 语言中，出于简化程序结构的考虑，不支持类间的多重继承，而只支持单重继承，即一个类至多只能有一个直接父类。接口是用来实现类间多重继承功能的结构。然而，接口的实现功能比多重继承更强。接口把方法的定义和类的层次区分开来，通过它可以在运行时动态地定位所调用的方法；同时，也可以实现“多重继承”，且一个类可以实现多个接口。正是这些机制，使得接口提供了比多重继承更简单、更灵活，而且更强健的功能。

12.7.1 接口的声明

接口的定义格式为：

```
interfaceDeclaration
{
    interfaceBody
}
```

其中，interfaceDeclaration 为接口声明部分，interfaceBody 为接口体部分。

1. 接口声明部分

```
[public] interface InterfaceName [extends superInterfaceList]
{
    ...
}
```

其中：

public 是访问控制修饰符，指明任意类均可以使用这个接口。在默认情况下，只有与该接口定义在同一个包中的类，才可以访问这个接口。

extends 子句与类声明中的 **extends** 子句基本相同，不同的是，一个接口可以有多个父接口，用逗号隔开，而一个类只能有一个父类。子接口继承父接口中所有的常量和方法。

2. 接口体的定义

接口体中包括常量定义和方法定义，其格式如下：

```
type constantName = Value;
returnType methodName([paramList]);
```

在接口中定义的常量可以被用来实现该接口的多个类共享，与 C 语言中的 **const** 定义常量是相似的。在接口中定义的常量必须是 **public static final**，这是系统默认的规定，所以常量也可以没有任何修饰符。

接口中只有方法声明，而无方法实现，所以，方法定义没有方法体，且用分号作为结束。接口中声明的方法必须是 **public abstract**。如果方法体是用其他语言编写的，则该接口方法可以用 **native** 修饰符修饰。

另外，如果在子接口中定义了与父接口同名的常量或相同的方法，则父接口中的常量被隐藏，方法被覆盖。

例如，下列定义的接口：

```
interface Collection
{
    int MAX_NUM=100;
    void add (Object objAdd);
    void delete (Object objDelet);
```



```
Object find (Object objFind);  
int currentCount();  
}
```

接口定义中声明了一个常量和 4 个方法。这个接口可以由队列、堆栈、链表等来实现。

12.7.2 接口的实现

接口的声明仅仅给出了抽象方法，要具体地实现接口所规定的功能，则需某个类为接口中的抽象方法定义实在的方法体，这就称为接口的实现。

在类的声明中，用 **implements** 子句表示一个类将要实现某个接口，在类体中可以引用接口中定义的常量，而且必须实现接口中定义的所有方法。一个类可以实现多个接口，在 **implements** 子句中用逗号分隔。

下面的程序在类 **FIFOQueue** 中实现了 12.7.1 节中所定义的接口：

```
class FIFOQueue implements collection  
{  
    public void add (Object objAdd)  
    { ... }  
    public void delete (Object objDelet)  
    { ... }  
    public Object find (Object objFind)  
    { ... }  
    public int currentCount()  
    { ... }  
}
```

注意：在类中实现接口所定义的方法时，方法的声明必须与接口中所定义的完全一致。在类中实现接口所定义的方法时，必须显式地使用 **public** 修饰符，否则将被系统警告为缩小了接口中定义的方法的访问控制范围。抽象类可以不实现接口的抽象方法，而非抽象类必须实现接口中的所有方法。

12.7.3 接口的回调

可以把实现某一接口类创建的对象引用赋给该接口声明的接口变量，之后，接口变量就可以调用被类实现的接口中的方法。

【例 12-11】HuiDiao.java 源程序。

```
class HuiDiao  
{  
    public static void main(String args[])  
    {
```



```

        ShowBrand sb;           //接口变量
        sb=new TV();
        sb.show("ChangHong TV");
        sb=new PC();
        sb.show("IBM PC");
    }//end of main method
}//end of class
interface ShowBrand
{
    void show(String s);
}
class TV implements ShowBrand
{
    public void show(String s)
    {
        System.out.println("*"+s);
    }
}
class PC implements ShowBrand
{
    public void show(String s)
    {
        System.out.println("#"+s);
    }
}

```

运行结果为:

```

*ChangHong TV
#IBM PC

```

12.7.4 接口作为参数

可以把实现某一接口类创建的对象引用赋给该接口声明的接口变量,之后,接口变量就可以调用被类实现的接口中的方法。

【例 12-12】InterPara.java 源文件。

```

interface SpeakHello
{
    void speakHello();
}

```



```
class Chinese implements SpeakHello
{
    public void speakHello()
    {
        System.out.println("中国人的习惯用语：你好，吃饭了吗？");
    }
}

class English implements SpeakHello
{
    public void speakHello()
    {
        System.out.println("英国人的习惯用语：你好，天气不错！");
    }
}

class KindHello
{
    public void lookHello(SpeakHello hello)
    {
        hello.speakHello();
    }
}

public class InterPara
{
    public static void main(String args[])
    {
        KindHello kindHello=new KindHello();
        kindHello.lookHello(new Chinese());
        kindHello.lookHello(new English());
    }
}
```



12.8 包

利用 Java 语言开发实际系统时，可以利用包（Package）来管理类。包是松散的类的集合，为了方便编程和管理，通常把需要在一起工作的类放在一个包里。

用 Java 语言创建的一个源码文件时，称为一个“编辑单元”，每个编辑单元的文件名必须以.java 为后缀，在编辑单元的内部，必须有一个与文件同名的 public（或非 public）主类。Public 只能用于修饰主类，其他类不能加 public 修饰符。

12.8.1 创建包

在默认情况下，系统为每个.java 源文件创建一个无名包，该文件中定义的所有类都隶属于该包，但由于该包无名字，所以不能被其他包引用。为了解决这个问题，就要创建有名包。

用 `package` 关键字创建包，而且该语句是.java 源文件的第一条语句：

```
package 包名;
```

例如：

```
package CardClass;

package CardSystem.CardClass;
```

实际上，创建包就是在当前目录下创建一个子目录，以便存放这个包中包含的所有类的.class 文件。上面的第二条语句中的“.”代表了目录分割符，即包中的类放在当前目录的 CardSystem 子目录的 CardClass 子目录下。

12.8.2 包的引用

在默认情况下，一个类只能引用同一个包中的类。如果需要使用其他包中的 `public` 类，可使用如下的方法。

1. 使用包名、类名前缀

对于同一个包中的其他类，只需在要使用的属性或方法名前加上类名作为前缀即可；对于其他包中的类，则需要在类名前缀的前面再加上包名前缀。

例如：

```
CardClass.P200_Card my200 = new CardClass.P200_Card();
System.out.println(my200.toString());
```

2. import 语句

加前缀的方法使用起来非常麻烦，可以使用 `import` 语句加载需要使用的类或包。

例如，加载类：

```
import CardClass.P200_Card;

P200_Card my200 = new CardClass.P200_Card();
```

例如，加载包：

```
import CardClass.*;

import java.awt.*;
```

3. CLASSPATH

环境变量 `CLASSPATH` 类似于 DOS 下的 `PATH`，它指明了所有默认的类型字节码文件路径。系统会自动在 `CLASSPATH` 中指明的路径下去寻找相应的类：

```
CLASSPATH=.;C:\Program Files\Java\jre1.6.0_06\lib\rt.jar;D:\java_class;
```

例如，有一个类是 `chen.Rose`，编译好后怎么存放？我们可以在 D 盘的 `java_class` 目录



下建立一个 `chen` 子目录，然后把 `Rose.class` 文件复制到该子目录下：

```
D:\java_class\chen\Rose.class
```

若需引用：

```
import chen.Rose;
```

或

```
import chen.*;
```

一些公司经常把它们的一组类打包发行，把这个包压缩成 `Jar` 或 `ZIP` 文件，设置的方法如下：

```
set CLASSPATH="d:\java_class; d:\oracle\ora.jar"
```

只需把文件名包含到 `CLASSPATH` 中，就可以正确地引用该压缩包中的类。



12.9 数组

在 `Java` 中，数组是一种专门的类型，它是有序数据的集合，数组中的每个元素的数据类型都是相同的。对元素的确定是由数组名和它的下标实现的，例如，`a[0]`代表数组 `a` 的第一个元素，`a[1]`代表数组 `a` 的第二个元素，依次类推。

12.9.1 数组声明

1. 数组的声明

`Java` 语言的数组声明采用与 `C` 语言类似的形式。数组可分为一维数组和多维数组。它们的声明形式为：

```
type arrayName[][][...];
```

也可使用另一等价形式：

```
type[][][...] arrayName;
```

其中，`type` 是 `Java` 语言支持的任意数据类型；`arrayName` 为数组名，它是一个合法的 `Java` 标识符；`[]`在这里表示是数组，而不是表示可选；`[][]...`表示是数组的方括号对，在这里可以多次重复，即方括号对可以有多个。它们的数量，表明数组维数。

例如：

```
int count[];           //一维整型数组 count
char ch[][];           //二维字符型数组 ch
float[] fNum;          //一维浮点型数组 fNum
```

对于 `Java` 编程人员，可能更喜欢采用后一种数组声明形式。它体现了 `Java` 语言的纯面向对象对象的特征。如下例所示：

```
int[] TestArray(int arraySize)
{
    int[] aId = new int [arraySize];
    return aId;
}
```

2. 创建数组空间

在 Java 数组声明中，不需要指明数组大小，这是因为数组声明并未为数组元素分配存储单元。要为数组元素分配存储单元，必须显式使用 `new` 运算符实现。其格式如下：

```
arrayName[][][...]= new type [Size1][[ Size2]...];
```

或

```
arrayName = new type [Size1][[ Size2]...];
```

其中，`arraySize1,array Size2...` 为分配给相对应的维的大小，经 `new` 运算符分配存储空间后，就可以引用数组中的元素了。

声明数组与为数组分配存储单元两部分，可以合在一起，格式如下：

```
type arrayName[][][...]= new type [Size1][[Size2]...];
```

或

```
type arrayName = new type [Size1][[Size2]...];
```

例如：

```
int count[] = new int [10];
```

```
char ch[][] = new char [3][5];
```

```
float fNum = new float [20];
```

12.9.2 数组元素的引用及初始化

1. 数组元素的引用

对已经分配了存储空间的数组（由 `new` 运算符实现），就可以引用数组中的每一个元素了。下标放在数组名后面的 `[]` 中，通过对下标的操作来引用元素。我们可以赋值给元素，也可以使用元素的值。数组元素引用的形式为：

```
arrayName[index1][[]...]
```

其中，`index1` 等为数组的下标。因为数组元素下标是从 0 开始的，所以最后一个元素的下标为 `arraySize-1`。对于每一个数组，都有一个属性 `length`，来指明数组的长度。

例如：

```
int count[] = new int [10];
```

```
char ch[][] = new char [3][5];
```

```
float fNum = new float [20];
```

数组 `count` 的元素为：`count[0]`，`count[1]`，`...`，`count[9]`。

数组 `ch` 的元素为：`ch[0][0]`，`ch[0][1]`，`...`，`ch[2][4]`。

数组 `fNum` 的元素为：`fNum[0]`，`fNum[1]`，`...`，`fNum[19]`。

前两个数组的属性分别为：

```
count.length=10
```

```
ch.length=15
```



2. 数组元素的初始化

在声明数组时，也可以同时对数组进行初始化。它的一维形式如下：

```
type arrayName[] = { Value1, Value2, ..., ValueN};
```

例如：

```
float myf[] = {1,2,3,4,5,6}
int myint[][]={{1,2,3},{4,5,6}};
//int myint[][]=new myint[2][3];
```

与 C 语言或 C++ 语言不一样，Java 语言要对数组元素的下标进行越界检查，以确保数据的安全。

初始化数组元素为对象的数组，假设 P200_Card 的构造函数为：

```
P200_Card (long cn,int pw,double b,double a)
{
    cardNumber = cn;
    password = pw;
    balance = b;
    additoryFee = a;
}
```

可用专门的语句来完成初始化工作：

```
P200_Card[] = Array200Card = new P200_Card[10];
for(int i=0; i< Array200Card.length; i++)
{
    Array200Card[i] = new P200_Card(20011019000+i,1234,50,0.1);
}
```

【例 12-13】 给数组赋值后，再将其打印出来。

```
public class ArrayRefer
{
    public static void main(String args[])
    {
        int IdA;
        int arrayA[] = new int [5];
        for (int i=0; i<5; i++)
            arrayA[i] = i;
        for (int i=0; i<arrayA.length; i++)
            System.out.println("arrayA[" + i + "] = " + arrayA[i]);
    }
}
```

运行结果为：

```
arrayA[0] = 0
```

```

arrayA[1] = 1
arrayA[2] = 2
arrayA[3] = 3
arrayA[4] = 4

```

【例 12-14】求 Fibonacci 数列，可表达为： $F[1]=F[2]=1$ ， $F[n]=F[n-1]+F[n-2]$ ，其中， $n \geq 3$ 。

```

public class Fibonacci
{
    public static void main(String args[])
    {
        int NumI;
        int Fib[] = new int [10];
        Fib[0]=1;
        Fib[1]=1;
        for (int i=2; i<10; i++)
            Fib[i]=Fib[i-1]+Fib[i-2];
        for (int i=1; i<=Fib.length; i++)
            System.out.println("Fib["+i+"] = "+Fib[i-1]);
    }
}

```

运行结果为：

```

Fib[1] = 1
Fib[2] = 1
Fib[3] = 2
Fib[4] = 3
Fib[5] = 5
Fib[6] = 8
Fib[7] = 13
Fib[8] = 21
Fib[9] = 34
Fib[10] = 55

```



12.10 字符串

字符串是编程中经常要用到的数据结构，它是字符的序列，从某种程度上来说，类似于字符的数组。实际上，在 C 语言中，字符串是用字符数组来实现的。而在面向对象的 Java 语言中，字符串是用类来实现的。

程序中用到的字符串可分成两大类：字符串常量和字符串变量。在 Java 中存放字符串常量的对象属于 String 类；对于字符串变量，由于程序经常需要对其进行添加、插入、修改之类的操作，所以一般都存放在 StringBuffer 类的对象中。



12.10.1 String类

字符串常量用 **String** 类的对象表示。字符常量是用单引号括起来的单个字符，例如，'a'、'\n'等。字符串常量是用双引号括起来的字符序列，例如，"a"、"/n"、"Hello"等。

C 语言中的字符串是由数组组成的，每个字符串的末尾以“\0”标志；而 Java 语言中的字符串常量，通常作为 **String** 类的对象存在，有专门的属性来规定它的长度。对于所有用双引号括起来的字符串常量，系统都会为它创建一个无名的 **String** 类型对象。

1. 创建String对象

String 类的构造函数及其使用方法如下。

public String(): 用来创建一个空的字符串变量。

public String(String value): 利用已存在的字符串常量创建一个新的 **String** 对象，可以用双引号括起来的直接常量。

public String(StringBuffer buffer): 利用已存在的 **StringBuffer** 对象对新创建 **String** 对象进行初始化。

public String(char value[]): 利用已存在的字符数组的内容初始化新创建的 **String** 对象。

下面是创建 **String** 对象的例子：

```
String s;  
//声明一个 String 对象，此时 s 的值为空  
s = new String("abc");  
//为 s 开辟内存空间，并初始化  
String s = new String("abc");  
//把上两条语句的功能合而为一  
String s = "abc";  
//这里的赋值是一种特殊的省略写法，Java 系统会自动为用双引号括起来的字符串常量  
//创建一个 String 对象，所以该语句的实际效果同上
```

2. 求字符串常量的长度

```
public int length();
```

例如：

```
String s = "Hello!";  
System.out.println(s.length());  
String t = "你过得可好? ";  
System.out.println(t.length());
```

运行结果为：

```
6  
6
```


3. 判断字符串的前缀、后缀

```
public boolean startsWith(String prefix);
```

```
public boolean endsWith(String suffix);
```

例如，成都地区的邮政编码以 61 开头：

```
if(s.startsWith("61"))
```

```
System.out.println("成都地区");
```

又如，旧的居民身份证号码最后 1 位代表性别：

```
if(s.endsWith("0")|| s.endsWith("2")||s.endsWith("4")
```

```
|| s.endsWith("6")|| s.endsWith("8"));
```

```
System.out.println("此人是女性！");
```

4. 字符串中单个字符的查找

```
public int indexOf(int ch);
```

```
public int indexOf(int ch,int fromIndex);
```

方法 1：查找字符 `ch` 在当前字符串中第一次出现的位置，如果找不到，则返回-1。方

法 2：从在当前字符串的第 `fromIndex` 位字符之后，开始查找字符 `ch` 第一次出现的位置，如果找不到，则返回-1。

例如：

```
String s = "Java 是面向对象的语言，JavaScript 是脚本语言。";
```

```
int i = s.indexOf((int)'J');
```

```
System.out.println(i);
```

```
int j = s.indexOf((int)'J',6);
```

```
System.out.println(j);
```

运行结果为：

```
0
```

```
13
```

下面两个方法与上面的方法类似，不同之处是从尾部向前开始查找：

```
public int lastIndexOf(int ch);
```

```
public int lastIndexOf(int ch,int fromIndex);
```

例如：

```
String s = "Java 是面向对象的语言，JavaScript 是脚本语言。";
```

```
int a = s.lastIndexOf((int)'J');
```

```
System.out.println(a);
```

```
int b = s.lastIndexOf((int)'J',10);
```

```
System.out.println(b);
```

运行结果为：

```
13
```

```
0
```



5. 字符串中子串的查找

```
public int indexOf(String str);
public int indexOf(String str,int fromIndex);
public int lastIndexOf(String str);
public int lastIndexOf(String str,int fromIndex);
public char charAt(int index);//获取当前字符串中第 index 位的字符
```

例如:

```
String s = "Java 是面向对象的语言, JavaScript 是脚本语言。";
int i = s.indexOf("语言");
System.out.println(i);
int j = s.indexOf("语言",12);
System.out.println(j);
char c = s.charAt(5);
System.out.println(c);
```

运行结果为:

```
10
26
面
```

6. 比较两个字符串

```
public int compareTo(String anotherString);
public boolean equals(Object anoObject);
public boolean equalsIgnoreCase(String anotherString);
```

方法 1: 如果当前字符串与参数字符串完全相同, 则 `compareTo()` 方法返回 0; 如果当前字符串按字母序大于参数字符串, 则 `compareTo()` 方法返回大于 0 的整数; 反之, 则返回小于 0 的整数。

方法 2: `equals` 重载 `Object` 类的方法, 如果当前字符串与参数字符串完全相同, 则返回真; 否则返回假。

方法 3: `equalsIgnoreCase` 与方法 `equals` 的用法相似, 不同之处是, 它比较时不计大小写。

例如:

```
String s1 = "Hello! ";
String s2 = "hello! ";
boolean b1 = s1.equals(s2);
boolean b2 = s1.equalsIgnoreCase(s2);
System.out.println(b1);
System.out.println(b2);
String s = "abc",s3 = "aab",s4 = "abd",s5 = "abc";
```

```
int i,j,k;
i = s.compareTo(s3);
j = s.compareTo(s4);
k = s.compareTo(s5);
System.out.println(i);
System.out.println(j);
System.out.println(k);
```

运行结果为：

```
false
true
1
-1
0
```

7. 连接字符串

```
public String concat(String str);
```

该方法把参数字符串连接到当前字符串的尾部，并返回这个连接而成的字符串，但当前字符串本身不改变。

例如：

```
String s = "Hello ";
System.out.println(s.concat("World! "));
System.out.println(s);
```

运行结果为：

```
Hello World!
Hello
```

12.10.2 StringBuffer类

StringBuffer 类的每个对象都是可以扩充和修改的字符串变量。

1. 创建StringBuffer对象

StringBuffer 类的构造函数如下：

```
public StringBuffer();
public StringBuffer(int length);
public StringBuffer(String str);
```

第一个函数创建了一个空的 StringBuffer 对象，第二个函数给出了新建的 StringBuffer 对象的长度，第三个函数利用一个已经存在的字符串 String 对象来初始化 StringBuffer 对象。



例如：

```
StringBuffer MyStrBuff1 =new StringBuffer();  
StringBuffer MyStrBuff2 =new StringBuffer(5);  
StringBuffer MyStrBuff3 =new StringBuffer("Hello!");
```

2. 字符串变量的扩充、修改和操作

StringBuffer 类有两组用来扩充其中所包含的字符的方法：

```
public StringBuffer.append(参数类型 参数名);  
public StringBuffer.insert(int 插入位置,参数类型 参数名);
```

例如：

```
StringBuffer MyStrBuff = new StringBuffer();  
MyStrBuff.append("Hello, Guys!");  
System.out.println(MyStrBuff.toString());  
MyStrBuff.insert(6,30);  
System.out.println(MyStrBuff.toString());
```

运行结果为：

```
Hello, Guys!  
Hello,30 Guys!
```

注意：println 方法不接受 **StringBuffer** 类型的参数，若希望在屏幕上显示出来，必须先调用 **toString** 方法把它转换成字符串常量。

StringBuffer 用来修改字符串的方法：

```
public void setCharAt(int index, char ch);
```

例如：

```
StringBuffer MyStrBuff = new StringBuffer("goat");  
System.out.println(MyStrBuff.toString());  
MyStrBuff.setCharAt(0, 'c ');  
System.out.println(MyStrBuff.toString());
```

运行结果为：

```
goat  
coat
```

3. 字符串的赋值和加法

字符串是经常使用的数据类型，为了编程方便，Java 编译系统中引入了字符串的加法和赋值。

例如：

```
String MyStr = "Hello,";  
MyStr = MyStr + "Guys!";
```

这两条语句乍看似乎有问题，因为 **String** 是字符串常量，实际上，它们相当于：

```
String MyStr = new StringBuffer().append("Hello,").toString();
MyStr = new StringBuffer().append(MyStr).append("Guys!").toString();
```

12.10.3 Java Application 命令行参数

Java Application 程序中用 `main()` 方法中的参数 `args[]` 来接受命令行参数。该参数是一个字符串数组，每个数组元素保存一个输入的命令行参数。

【例 12-15】UsePara.java 源程序。

```
public class UsePara
{
    int a0,a1,a2;
    if(args.length!=2)
    {
        System.out.println("运行本程序应提供两个整数命令行参数");
        System.exit(0);
    }
    a0 = Integer.parseInt(args[0]);
    a1 = Integer.parseInt(args[1]);
    //Integer 数据类型类的 parseInt 方法把字符串型转换为整型数
    a2 = a0 * a1;
    System.out.println(a0+"*"+a1+"="+a2);
}
```

运行程序：

```
java UsePara 5 6
```

运行结果为：

```
5*6=30
```



12.11 语言基础类库

12.11.1 Object 类

Object 类是 Java 中所有类的直接或间接父类。

它的主要方法如下：

```
protected Object clone();
//生成当前对象的一个副本，并返回这个复制对象
public boolean equals(Object obj);
//比较两个对象是否相同，是则返回 true
```



```
public final Class getClass();  
//获取当前对象所属的类信息，返回 Class 对象  
protected void finalize();  
//定义回收当前对象所需完成的清理工作  
public String toString();  
//返回当前对象的有关信息
```

12.11.2 数据类型类

Java 中，数据类型类与对应的基本数据类型参见表 12-1。

表 12-1 数据类型类与对应的基本数据类型

数据类型类	基本数据类型
Boolean	boolean
Character	char
Double	double
Float	float
Integer	int
Long	long

下面以 Integer 类为例介绍数据类型类的用法。
Integer 类的域：MAX_VALUE 域和 MIN_VALUE 域，规定了 int 类型的最大值和最小值。
构造函数：

```
public Integer(int value);  
public Integer(String s);
```

数据类型转换：

```
public double doubleValue();  
public int intValue();  
public long longValue();  
public String toString(); //把当前对象所对应的 int 值转化为字符串  
public static int parseInt(String s); //可方便地把字符串转换为 int 值
```

例如：int i = Integer.parseInt("250");

其他字符串转换方法：

```
public static Integer valueOf(String s);
```

例如：int i = Integer.valueOf("250").intValue();

对于其他数据类型，只需替换类名，用法类似。下例中，把 Int 换成 Float 或 Long 即可。

```
float f = Float.parseFloat("25.0");  
double d = Double.parseDouble("25.7890");  
long l = Long.parseLong("25");
```

12.11.3 Math类

Math 类的主要方法如下：

<code>public final static double E;</code>	//数学常量 e
<code>public final static double PI;</code>	//圆周率常量
<code>public static double abs(double a);</code>	//绝对值
<code>public static double acos(double a);</code>	//反余弦
<code>public static double exp(double a);</code>	//e 的参数次幂
<code>public static double log(double a);</code>	//自然对数
<code>public static double random();</code>	//产生 0~1 之间的伪随机数
<code>public static double pow(double a, double a);</code>	//乘方
<code>public static double sqrt(double a);</code>	//平方
<code>public static double rint(double a);</code>	//四舍五入

12.11.4 System类

System 不能实例化，所以它的所有方法和属性都是 static 的。

它有 3 个属性：

```
public static PrintStream err; //标准错误输出
public static InputStream in; //标准输入
public static PrintStream out; //标准输出
```

例如：

```
char c = System.in.read();
System.out.println("Hello!");
```

常用方法：

```
public static void exit(int status);
//强制退出运行状态，并把状态信息返回给运行虚拟机的操作系统。
```

例如：

```
System.exit(0);
```



12.12 Applet类与Applet程序

12.12.1 Applet类

Applet 类存在于 java.applet 包中，是 java.awt.Panel 的子类。Panel 是 Container 的一种，它可以：包容和排列其他的界面元素，如按钮、对话框和其他容器；响应它所包容范围之内



的事件, 或把事件向更高层次传递。Applet 在拥有上述作用的基础上, 还具有一些与浏览器和 Applet 生命周期有关的方法。图 12-3 显示了 Applet 的生命周期。

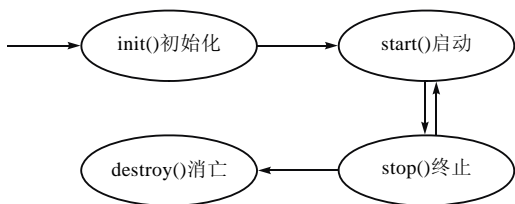


图 12-3 Applet 的生命周期

用户自定义的 Applet 子类是 Java Applet 程序的标志。在实际运行中, 浏览器在下载字节码的同时, 会自动创建一个用户定义的 Applet 子类的实例, 并在适当的时候自动调用相应的实例方法。

init()方法: 完成实例的初始化工作。浏览器创建 Applet 子类的实例, 并调用该方法。

start()方法: 用来启动浏览器运行 Applet 的主线程。init()方法初始化后调用 start()方法。包含 Applet 的 HTML 页面被重新加载时, 其中的 Applet 实例会被重新启动并调用 start()方法, 但 init()方法只被调用一次。

paint()方法: 主要作用是在 Applet 的界面中显示文字、图形和其他界面元素。paint()方法有一个固定参数: Graphics 类的对象 g。当 Applet 子类的实例被初始化并启动时, 浏览器将自动生成一个 Graphics 类的实例 g, 并将 g 作为参数传给 Applet 实例的 paint()方法。导致浏览器调用 paint()方法的事件: Applet 被启动后, 自动调用 paint()方法重新绘制自己的界面; Applet 所在的浏览器窗口被改变, 例如, 窗口被放大、缩小、移动, 或被其他窗口遮挡又重新显示在最前方等, 都需重绘界面; 其他相关方法被调用, 也会相应地调用 paint()方法。

stop()方法: 类似于 start()方法的逆操作。当用户浏览其他 WWW 页面或切换到其他应用时, 浏览器将暂停执行 Applet 的主线程, 在暂停之前会调用 stop()方法。

destroy()方法: 当用户退出浏览器时, Applet 实例也被消灭, 消灭之前调用 destroy()方法完成一些释放资源、关闭连接之类的操作。

【例 12-16】 本例用方块图显示 Applet 各个方法被调用的次数, 下面是 LifeCycle.java 源文件和嵌入 Applet 的 life.html 文件。

LifeCycle.java 源文件:

```
import java.awt.*;
import java.applet.*;
public class LifeCycle extends Applet
{
    private int initCount;
    private int startCount;
    private int stopCount;
    private int destroyCount;
    private int paintCount;
    public LifeCycle()
    {
        initCount=0;
        startCount=0;
```



```
        stopCount=0;
        destroyCount=0;
        paintCount=0;
    }
    public void init()
    {
        initCount++;
    }
    public void destroy()
    {
        destroyCount++;
    }
    public void start()
    {
        startCount++;
    }
    public void stop()
    {
        stopCount++;
    }
    public void paint(Graphics g)
    {
        paintCount++;
        g.drawLine(20,200,300,200);//横坐标
        g.drawLine(20,200,20,20);//纵坐标
        g.drawLine(20,170,15,170);//标尺
        g.drawLine(20,140,15,140);
        g.drawLine(20,110,15,110);
        g.drawLine(20,80,15,80);
        g.drawLine(20,50,15,50);
        g.drawString("init()",25,213);
        g.drawString("start()",75,213);
        g.drawString("stop()",125,213);
        g.drawString("destroy()",175,213);
        g.drawString("paint()",225,213);
        g.fillRect(25,200-initCount*30,40,initCount*30);
        g.fillRect(75,200-startCount*30,40,startCount*30);
        g.fillRect(125,200-stopCount*30,40,stopCount*30);
    }
}
```



```

g.fillRect(175,200-destroyCount*30,40,destroyCount*30);
g.fillRect(225,200-paintCount*30,40,paintCount*30);
}
}

```

life.html 文件:

```

<HTML>
<HEAD>
<TITLE>图形用户界面 </TITLE>
</HEAD>
<BODY>
<APPLET CODE="LifeCycle.class" WIDTH=500 HEIGHT=500>
</APPLET>
</BODY>
</HTML>

```

运行结果如图 12-4 所示。

12.12.2 HTML文件的参数传递

在<APPLET> 和</APPLET>之间嵌入 Applet 标记, 常用参数: code, height 和 width。

一些可选参数如下。

codebase: 当 Applet 字节码文件和它所嵌入的 HTML 文档所在的目录不同时, 用 codebase 指明 Applet 所在的位置, 用 URL 格式表示, 例如,

```
codebase=http://www.illusion.org/Applet/MyApplet.class
```

alt: 如果浏览器不支持 Java, 无法执行字节码文件, 就把 alt 参数指明的信息显示给用户, 例如,

```
alt="This is a Java Applet ,your browser cant not understand"
```

align: 指明 Applet 界面区域在浏览器窗口中的位置情况, 例如,

```
align=CENTER
```

<PARAM>标记: HTML 文件可向它所嵌入的 Applet 传递参数, 每个标记只能传递一个字符串型的参数。

【例 12-17】 MyAppletParam.java 可以接收 param.html 文件中传递的参数。

param.html 文件:

```

<HTML>
<BODY>
<APPLET code="MyAppletParam" height=200 width=300>
<PARAM name=vstring value="我是来自 HTML 的参数">
<PARAM name=x value=50>

```

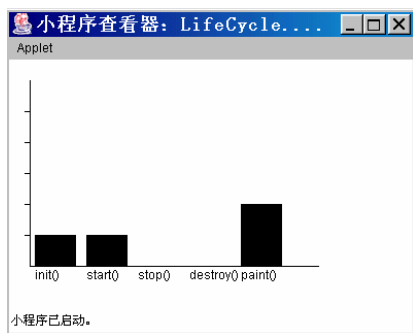


图 12-4 运行结果

```
<PARAM name=y value=100>
```

```
</BODY>
```

```
</HTML>
```

MyAppletParam.java 文件:

```
import java.awt.*;
```

```
import java.applet.*;
```

```
public class MyAppletParam extends Applet
```

```
{
```

```
    private String s="";
```

```
    private int x;
```

```
    private int y;
```

```
    public void init()
```

```
    {
```

```
        s=getParameter("vstring");
```

```
        x=Integer.parseInt(getParameter("x"));
```

```
        y=Integer.parseInt(getParameter("y"));
```

```
    }
```

```
    public void paint(Graphics g)
```

```
    {
```

```
        if(s!=null)
```

```
            g.drawString(s,x,y);
```

```
    }
```

```
}
```

运行结果如图 12-5 所示。



12.13 异常处理

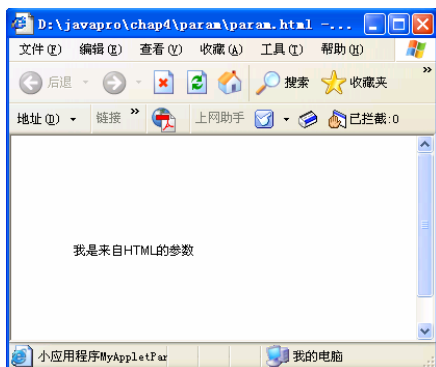
异常 (Exception) 又称例外, 是指程序执行过程中出现的不正常现象。任何一个程序都可能出现异常, 例如, 要打开的文件不存在、内存不够、数组访问越界等。Java 作为一种面向对象语言, 它设计了很多标准的异常处理类, 采用 throw-catch (抛出一捕获) 方式, 捕获并处理异常。Java 语言使异常处理标准化, 使程序设计思路更清楚, 理解更容易。本章将介绍异常处理方法。

12.13.1 Java语言异常处理的特点

与传统的错误处理技术相比, Java 语言使用异常处理方法有以下优越性:

- ◎ 可以把错误处理代码和常规代码隔离开;
- ◎ 可以在调用栈中传播错误。

错误!





1. 隔离错误处理代码和常规代码

在传统的程序设计中，错误检测、报错和错误处理放在一起，因而经常导致代码产生许多复杂的分支。

下面用伪代码描述一个读文件的函数，代码如下：

```
read file
{
    open the file;
    determine its size;
    allocate memory;
    read file into memory;
    close the file;
}
```

这个函数看来简单，但它忽略了以下的潜在错误：

- ◎ 文件不能打开；
- ◎ 文件长度不能确定；
- ◎ 没有分配足够的内存；
- ◎ 读失败；
- ◎ 文件关闭失败。

要解决这些问题，函数中必须添加许多代码来进行错误检测、报错和错误处理，代码修改如下：

```
readFile
{
    initialize errorCode = 0;
    open the file;
    if(the file open)
    {
        determine its size;
        if(got the file length)
        {
            allocate memory;
            if(got Enough memory)
            {
                read file into memory;
                if(read failed)
                    errorCode = 1;    //读失败
            }
        }
        else
            errorCode = 2;    //内存分配失败
    }
}
```

```

    }
    else
        errorCode = 3;           //文件长度不能确定
    }
    close the file;
    if(the file did not close && errorCode = 0)
        errorCode = 4           //文件关闭失败
    return errorCode;
}

```

在增加了错误处理之后，使原先的代码变得很复杂。而 Java 的异常处理可解决该问题，代码如下：

```

readFile
{
    try
    {
        open the file;
        determine its size;
        allocate memory;
        read file into memory;
        close the file;
    }
    catch(file open failed)
    {
        dosomething;
    }
    catch(size determination failed)
    {
        dosomething;
    }
    catch(memory allocation failed)
    {
        dosomething;
    }
    catch(read failed)
    {
        dosomething;
    }
}

```



用户不需要在主程序中检测和处理错误,异常处理提供了一种当错误发生时分离所有处理细节和原有代码的方法。

2. 在调用栈中传播错误

假设上例中的 `readFile` 方法是从主程序的第 4 级调用的,即由 `method1` 调用 `method2`、由 `method2` 调用 `method3`、再由 `method3` 调用 `readFile`, 如下:

```
method1
{
    call method2;
}
method2
{
    call method3;
}
method3
{
    readFile;
}
```

进一步假设,只有 `method1` 对 `readFile` 中的错误感兴趣。传统的错误检测技术需要调用栈中的方法 `method2` 和方法 `method13` 来辅助错误代码的传播, `readFile` 返回的错误才能最终到达 `method1` 方法,其代码如下:

```
method1
{
    errorCodeType error;
    error = call method2;
    if(error)
        doErrorProcessing;
    else
        proceed;
}
errorCodeType method2
{
    errorCodeType error;
    error = call method3;
    if(error)
        return Error;
    else
        proceed;
}
```

```

errorCodeType method3
{
    errorCodeType error;
    error = call readFile;
    if(error)
        return Error;
    else
        proceed;
}

```

这样，在调用栈中的所有方法都需编写错误处理代码。而 Java 中的异常处理具有在方法调用栈中传播错误的功能，即可以向调用栈的上层方法传递异常，可以让调用栈的更上层的方法来捕获异常，做到只让关心错误的方法捕获它。

```

method1
{
    try
    {
        call method2;
    }
    catch
    {
        do error processing;
    }
}
method2 throws exception
{
    call method3;
}
method3 throws exception
{
    call readFile;
}

```

中间代码的工作就是抛出（throw）异常。

12.13.2 异常类的层次

Java 的异常类有自己的类层次，如图 12-6 所示。所有异常类都是 `Throwable` 类的子类，`Throwable` 属于 `java.lang` 包，在程序中不必使用 `import` 语句引入即可使用。`Exception` 和 `Error` 是 `Throwable` 类的直接子类。`Exception` 是程序中所有可能恢复的异常类的父类，



`RuntimeException` 和 `Non_RuntimeException` 是 `Exception` 的直接子类。`Error` 包含链接或装载时出现的错误及与虚处理机相关的错误,对于这一类非常严重的问题,一般不要求应用程序进行异常处理。

Java 程序可以抛出系统定义好的异常类,也可以由用户自定义新的异常类,用户定义的异常类一般是 `Throwable` 类或其子类。

1. 运行异常

运行异常是指 Java 程序在运行时发现的由 Java 运行系统引发的各种异常,其出现频率很高,检测运行异常的开销很大,所以编译器不要求捕获或声明运行异常。

常见的执行时由系统抛出的异常如下。

`ArithmeticException`: 算术运算中除数为 0,产生异常。

`ArrayIndexOutOfBoundsException`: 访问数组下标超界,产生异常。

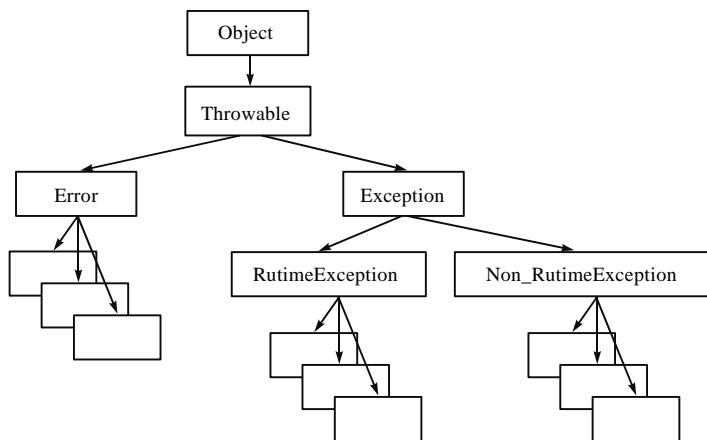


图 12-6 Java 的异常类

`ArrayStoreException`: 进行写数组操作时,对象或数据类型不兼容,产生异常。

`ClassCastException`: 当试图把对象 A 转换为对象 B 时,如果对象 A 既不是对象 B 的实例,又非对象 B 的子类,则产生异常。

`IllegalArgumentException`: 在方法的参数表中,如果参数无效,则产生异常。

`IllegalThreadStateException`: 非法改变线程状态,如启动已执行的线程,产生异常。

`NegativeArraySizeException`: 当数组的长度是负数时,产生异常。

`NullPointerException`: 试图访问空对象的变量、方法或空数组的元素,产生异常。

`SecurityException`: Applet 试图执行被 WWW 浏览器安全设置所禁止的操作,产生异常。

`IncompatibleClassChangeException`: 若类的定义被改变,而引用该类的其他类没有被重新编译,则产生异常。

`OutOfMemoryException`: “内存不足”异常。

`NoClassDefException`: Java 执行时找不到所引用的类,产生异常。

`IncompatibleTypeException`: 试图实例化一个接口,产生异常。

`UnsatisfiedLinkException`: 所调用的方法是本地方法,但执行时无法连接这个函数,将产生异常。

InternalException: 系统内部故障所导致的异常。

2. 非运行异常

非运行异常能在编译时检测到，是类 `Non_RuntimeException` 及其子类的实例，可通过 `throws` 语句抛出。Java 在其标准包 `java.lang`、`java.util`、`java.io`、`java.net` 中定义的异常类都是非运行异常类。

(1) `java.lang` 包中定义的异常

ClassNotFoundException: 找不到类和界面。

CloneNotSupportedException: 对象引用了 `Object` 类的 `Clone` 方法，但没有实现 `Cloneable` 界面。

IllegalAccessException: 试图去访问另一个包中的类的非 `public` 方法。

InstantiationException: 使用 `Class` 类的 `newInstance` 方法试图去创建类的实例时，由于指定的类为界面、抽象类、数组而不能创建。

InterruptedException: 当一个线程正在等待时，另一个线程来中断这个线程。

(2) `java.io` 包中定义的异常

java.io.EOFException: 在正常的输入操作完成之前遇到了文件结束。

java.io.FileNotFoundException: 找不到指定的文件。

java.io.UTFDataFormatException: 不能完成 Java 定义的 UTF 格式的字符串转换。

(3) `java.net` 包中定义的异常

java.net.ProtocolException: 网络协议有错。

java.net.SocketException: 不能正常完成 `socket` 操作。

java.net.UnknownHostException: 网络 `host` 名不能被解析为网络地址。

java.net.UnknownServiceException: 网络连接不能支持请求服务。

12.13.3 抛出异常

Java 通过使用 `throws` 语句指定异常类型并抛出异常，其格式如下：

```
public void readFile() throws IOException
{
    ...//包含可能产生 IO 异常的语句
}

public void myfunc() throws MyException1, MyException2
{
    ...
}
```

在 `throws` 语句中指定的异常类可以是实际抛出异常类型的父类。确定需要抛出哪些异常的方法有两种：一是通过经验，仔细查找所调用的方法可能产生的异常；二是先不做任何声明，等待编译器报错，编译器将通报所有需要用 `throws` 语句抛出的异常。

如果方法用 `throws` 语句声明异常，则意味着该方法不进行异常处理，仅仅抛出异常，



那么调用该方法的其他方法应该负责处理异常，这样有利于集中处理异常，进一步提高程序的可读性和可维护性。

12.13.4 异常处理

当程序发生异常时，就会抛出一个异常，这个异常可以被程序捕获，进行相应的处理。最基本的异常处理结构形式如下：

```
try
{
    正常程序段;
}
catch(异常类 1 异常变量)
{
    与异常类 1 有关的处理程序段;
}
catch(异常类 2 异常变量)
{
    与异常类 2 有关的处理程序段;
}
...
finally
{
    退出异常处理程序段;
}
```

在这个结构中使用了 3 个关键字：`try`、`catch` 和 `finally`。

`try` 后面大括号中的程序段称为保护代码（Protected Code），可能产生某种异常，在 `try` 的 `{}` 中不进行任何异常处理。紧接 `try` 之后的是一系列 `catch` 语句，这才是处理异常的正确地方。

`catch` 与方法的定义相似，需要一个参数，该参数必须是 `Throwable` 类或其子类的一个对象或接口。`catch` 的作用是，在系统或应用程序抛出一个异常后，与 `catch` 的参数进行比较，如果相匹配，就执行 `catch` 方法中的异常处理程序段，否则与下一 `catch` 继续进行相匹配的比较。

相匹配是指符合下列 3 种情况之一：

- ◎ 被抛出的异常类与 `catch` 参数指定类是同一类；
- ◎ 被抛出的异常类是 `catch` 参数指定类的子类；
- ◎ 如果 `catch` 参数是一个接口，而被抛出的异常类实现了这个接口。

`catch` 匹配检查是顺序进行的，当程序中包含多条 `catch` 语句时，如果 `catch` 的顺序安排不当，可能导致某些 `catch` 后的处理程序段永远不会被执行。例如下面的程序段：

```

catch (IOException e) {
    异常处理 1;
}

catch (FileNotFoundException e) {
    异常处理 2;
}

```

如果程序产生了一个 `FileNotFoundException` 异常，则在进行第一次匹配比较时，根据相匹配的第 2 种情况，Java 认为它们已经匹配，则执行异常处理 1，而异常处理 2 永远不会被执行。所以在使用 `catch` 语句时，应该清楚所有参数所属类的层次，以便正确安排 `catch` 语句的顺序。如果出现这种错误，`javac` 编译器将会发出错误信息：`catch not reached`。

无论 `try` 是否抛出异常，还是某个 `catch` 进行了异常处理，`finally` 后的程序段总要被执行一次。唯一不被执行的情况是：`try` 或 `catch` 中执行了 `System.exit()`，则程序立即中止。

在 `try-catch-finally` 这个编程结构中，`catch` 语句和 `finally` 语句是可选的，但至少使用其中一条语句。

12.13.5 嵌套的异常处理

在 `try-catch-finally` 结构中，可以使用嵌套形式，即在捕获异常处理过程中，可以继续抛出异常。在这种嵌套结构中，产生异常后，首先与最内层的 `try-catch-finally` 结构中的 `catch` 语句进行匹配比较。如果没有相匹配的 `catch` 语句，则该异常情况可以被抛出，让外层的 `try-catch-finally` 的结构重复进行匹配检查。这样从最内层到最外层，逐一检查匹配，直到找到一个匹配为止。如果所有的 `try-catch-finally` 结构中都没有找到一个与异常匹配的 `catch`，则 Java 系统会打印一个与该异常相关的信息并显示异常发生时的堆栈状态。



12.14 Java多线程机制

Java 语言从设计之初，就考虑到对多线程的支持，因而在系统级和语言级均提供了对多线程的支持。使用多线程的优点就是：当程序中有多个可执行线程时，能够充分利用系统资源，提高程序执行效率。本章介绍多线程应用程序的设计方法。

12.14.1 基本概念

1. 线程

线程是比进程更小的单位，是应用程序中的一个可执行线索，多线程就是同一个应用程序中有多个可执行线索，它们可以并发执行。

前面介绍的所有例程都是单线程的，即一个应用程序只存在一个可执行线索。在执行过程中，如果某部分代码因为等待某个 I/O 操作而受阻，则程序的其他部分即使与此无关也不能执行，这样就严重地浪费了 CPU 资源。多线程机制的提出就是为了解决这个问题。



回忆一下多进程并行执行的情况，在 CPU 上执行的某个进程因为等待某种资源而受阻时，多任务操作系统可以使该进程挂起（暂停执行），而根据某种调度原则启动另外一个不同的进程执行，直到前一进程获得其所需资源，才唤醒该进程，让它继续执行。这样，在多任务操作系统的调度下，可以让多个进程并行执行，能够较好地利用 CPU 资源，但仍然难以满足现代应用程序的需要。例如，需要在同一应用程序中完成声音播放、图像显示、网络文件下载等多项工作，如果使用传统的单线程程序，就只能顺序地逐一实现，而使用多线程方法则可以并发实现。

多线程就是同一程序中多个任务的并发实现。它与进程有相似之处，可以动态地生成或销毁等，它们之间的差别主要体现在以下两个方面：

- ◎ 作为基本的执行单元，线程的划分比进程小，因此，支持多线程的系统要比只支持多进程的并发程度高；
 - ◎ 进程把内存空间作为自己的资源之一，每个进程均有自己的内存单元，线程却共享内存单元，通过共享的内存空间来交换信息，从而有利于提高执行效率。
- 由于线程划分更小，涉及的资源更少，因而使用更加灵活、方便。

2. 线程调度与优先级

应用程序中的多个线程能够并发执行，但从系统的内部来看，所有线程仍然是串行的，一个一个地执行，那么，如何决定哪一个线程先执行，哪一个线程后执行呢？Java 引入了优先级的概念，优先级是指线程获得 CPU 而执行的优先程度，优先级越高，获得 CPU 的权力越大，执行的机会越多，执行的时间也越长。

Java 把优先级划分为 10 级，用 1~10 的整数表示，数值越大，优先级越高。在创建线程时，可以为线程分配优先级。在 Thread 类中定义了 3 个优先级常量：MIN_PRIORITY, MAX_PRIORITY 和 NORM_PRIORITY，其值分别为 1, 10, 5。在为线程分配优先级时，其值应该在 1~10 之间，否则将出错。如果应用程序没有为线程分配优先级，则 Java 系统为其赋值为 NORM_PRIORITY。

调度就是分配 CPU 资源，确定线程的执行顺序。Java 采用抢占式调度方式，即高优先级线程具有剥夺低优先级线程执行的权力。如果一个低优先线程正在执行，这时出现一个高优先级线程，那么低优先级线程就只能停止执行，放弃 CPU，退回到等待队列中，等待下一轮执行，而让高优先级线程立即执行。如果线程具有相同的优先级，则按“先来先服务”的原则调度。

理解了 Java 的抢占式调度策略后，在设计程序时，应该让高优先级线程执行一段时间后，能够交出使用权，放弃 CPU。有两种方法可以达到这一目的：

- ◎ 调用 sleep()方法，暂时进入睡眠状态，从而让出 CPU，使具有相同优先级线程和低优先级线程有执行的机会；
- ◎ 调用 yield()方法而放弃 CPU，这时，与它具有相同优先级的线程就有执行的机会。

3. 线程的状态与生命周期

每个 Java 程序都有一个默认的主线程：对于 Application，主线程是 main 方法执行的线索；对于 Applet，主线程指挥浏览器加载并执行 Java 小程序。要想实现多线程，必须在主

线程中创建新的线程对象。Java 语言使用 **Thread** 类及其子类的对象来表示线程。线程的生命周期如图 12-7 所示。

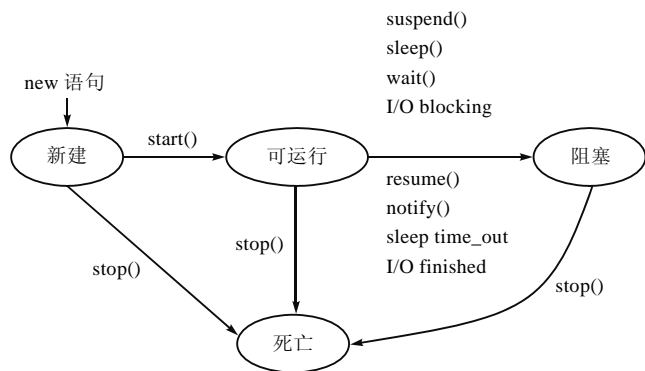


图 12-7 线程的生命周期

① 新建：当一个 **Thread** 类或其子类的对象被创建后，进入此状态。这时，线程对象已被分配内存空间，其私有数据已被初始化，但该线程还未被调度，可用 **start()** 方法调度，或者用 **stop()** 方法中止。新线程一旦被调度，就将切换到可执行状态。

② 可运行：处于可执行环境中，随时可以被调度而执行。它可细分为两个子状态：

◎ 运行状态，已获得 CPU，正在执行；

◎ 就绪状态，只等待处理器资源。

这两个子状态的过渡由执行调度器来控制。

③ 阻塞：由某种原因引起线程暂停执行的状态。

④ 死亡：当线程执行完毕或另一线程调用 **stop()** 方法使其停止时，进入这种停止状态，它表示线程已退出可运行状态，并且不再进入可运行状态。

4. 线程控制方法

Thread 类定义了许多控制线程执行的方法，它的定义如下：

```

public class Thread implements Runnable
{
    public final static int MIN_PRIORITY;
    public final static int NORM_PRIORITY;
    public final static int MAX_PRIORITY;
    public Thread();
    public Thread(Runnable target);
    public Thread(ThreadGroup group, Runnable target);
    public Thread(String name);
    public Thread(ThreadGroup group, String name);
    public Thread(Runnable target, String name);
    public Thread(ThreadGroup group, Runnable target, String name);
}
  
```



```
public void run();
public synchronized native void start();
public final void stop();
public final synchronized void stop(Throwable o);
public static native void yield();
public static native void sleep(long millis)
    throws InterruptedException;
public static void sleep(long millis, int nanos)
    throws InterruptedException;
public final void suspend();
public final void resume();
public final synchronized void join(long millis)
    throws InterruptedException;
public final synchronized void join(long millis, int nanos)
    throws InterruptedException;
public final void join() throws InterruptedException;
public void interrupt();
public static boolean interrupted();
public boolean isInterrupted();
public void destroy();
public final native boolean isAlive();
public final void setPriority(int newPriority);
public final int getPriority();
public final void setName(String name);
public final String getName();
public final ThreadGroup getThreadGroup();
public static native Thread currentThread();
public static int activeCount();
public static int enumerate(Thread tarray[]);
public native int countStackFrames();
public static void dumpStack();
public final void setDaemon(boolean on);
public final boolean isDaemon();
public void checkAccess();
public String toString();
}
```

程序中经常使用下面的方法，对线程进行控制。

start(): 用于调用 **run()**方法使线程开始执行。

stop(): 立即停止线程执行, 其内部状态清零, 放弃占用资源。

suspend(): 暂停线程执行。线程的所有状态和资源保持不变, 以后可以通过另一线程调用 **resume()** 方法来重新启动这个线程。

resume(): 恢复暂停的线程, 安排暂停线程执行。

sleep(): 调整 Java 执行时间, 所需参数是指定线程的睡眠时间, 以毫秒为单位。

join(): 调用线程等待本线程执行结束。

yield(): 暂停调度线程并将其放在等待队列末尾, 等待下一轮执行, 使同优先级的其他线程有机会执行。

12.14.2 多线程实现方法

创建新线程有两种方法:

- ◎ 生成 **Thread** 子类;
- ◎ 生成一个类, 声明实现 **Runnable** 接口。

无论采用哪种方法, 程序员可以控制的关键性操作有两个:

- ◎ 定义用户线程的操作, 即定义用户线程的 **run** 方法;
- ◎ 在适当时候建立用户线程实例。

1. 创建 Thread 类的子类

可以按以下步骤进行:

- ◎ 生成 **Thread** 类的子类;
- ◎ 在子类中覆盖 **run()** 方法;
- ◎ 生成子类的对象, 并且调用 **start()** 方法启动新线程。

以上这 3 个步骤可以用 Java 语言表述为:

```
class NewThread extends Thread
{
    ...

    public void run()
    { ... }
}
```

在需要创建 **NewThread** 这个线程的类或方法中, 生成 **NewThread** 对象:

```
NewThread thread = new NewThread();
thread.start();
```

经过这 3 步后, 新线程处于可执行状态。**start()** 方法将调用 **run()** 方法执行线程。**run()** 方法就是新线程完成具体工作的地方, **run()** 方法是 Java 执行时为了启动线程而调用的第一个用户定义方法, 就像 **main()** 是 Java 执行时为了启动 **Application** 程序而调用的第一个用户定义方法一样。

下面程序演示如何用生成 **Thread** 子类的方法来创建新线程。



【例 12-18】 ThreadTest1.java 源文件。

```
public class ThreadTest1
{
    public ThreadTest1()
    {
        FirstThread first = new FirstThread();
        SecondThread second = new SecondThread();
        first.start();
        second.start();
    }
    public static void main(String[] args)
    {
        new ThreadTest1();
    }
}

class FirstThread extends Thread
{
    public void run()
    {
        try
        {
            System.out.println("First thread starts running.");
            for(int i=0; i<10; i++)
            {
                System.out.println("First " + i);
                sleep(1000);
            }
            System.out.println("First thread finishes running.");
        }
        catch (InterruptedException e) {}
    }
}

class SecondThread extends Thread
{
    public void run()
    {
        try
        {
            System.out.println("\tSecond thread starts running.");
            for(int i=0; i<10; i++)
```



```

        {
            System.out.println("\tSecond " + i);
            sleep(1000);
        }
        System.out.println("\tSecond thread finishes running.");
    }
    catch (InterruptedException e) {}
}
}

```

程序设计了两个线程 `FirstThread` 类和 `SecondThread` 类，它们都是 `Thread` 类的子类覆盖了 `run()` 方法，在其中分别进行打印数值的工作。除了这两个线程外，还有一个线程在执行，就是启动类线程，称之为主线程，它负责生成另外两个线程，再用 `start()` 方法启动这两个线程。线程 `first` 和 `second` 启动后，并发执行。观察执行结果会发现，两个线程交替打印数据，而不是一个线程完成了所有打印工作后，另一个线程才开始打印工作，这就是多线程的本质。提示：线程在调用 `Thread` 类方法 `sleep()` 睡眠时，有可能产生异常，要求应用程序用 `try-catch` 捕获这个异常。如果不用 `try-catch`，程序将出错。

再看一个例程，它使用了 `Thread` 类的其他线程控制方法。

【例 12-19】`ThreadTest2.java` 源文件。

```

public class ThreadTest2
{
    public ThreadTest2()
    {
        FirstThread first = new FirstThread();
        SecondThread second = new SecondThread();
        first.start();
        second.start();
        try
        {
            System.out.println("Waiting for first thread to finish...");
            first.join();
            System.out.println("Waking up second thread...");
            second.resume();
        }
        catch (InterruptedException e) {}
    }
    public static void main(String[] args)
    {
        new ThreadTest2();
    }
}

```



```
}  
class FirstThread extends Thread  
{  
    public void run()  
    {  
        try  
        {  
            System.out.println("\tFirst thread STARTS running.");  
            for(int i=0; i<10; i++)  
            {  
                System.out.println("\tFirst " + i);  
                sleep(1000);  
            }  
            System.out.println("\tFirst thread FINISHES running.");  
        }  
        catch (InterruptedException e) {}  
    }  
}  
class SecondThread extends Thread  
{  
    public void run()  
    {  
        try  
        {  
            System.out.println("\t\tSecond thread STARTS running.");  
            for(int i=0; i<10; i++)  
            {  
                if(i== 4)  
                    suspend();  
                System.out.println("\t\tSecond " + i);  
                sleep(1000);  
            }  
            System.out.println("\t\tSecond thread FINISHES running.");  
        }  
        catch (InterruptedException e) {}  
    }  
}
```

程序仍然使用两个线程打印数据，不同的是，second 线程在打印数据过程中，当 i=4 时，

调用 `suspend()` 方法，暂停本身的执行。主线程用 `join()` 方法等待线程 `first` 执行结束后，用 `resume()` 方法来唤醒 `second` 线程，`second` 线程被唤醒后，将继续完成打印工作。提示：`join()` 方法也将出现 `InterruptedException` 异常，所以必须捕获异常。

2. 实现 `Runnable` 接口

使用这种方法创建新线程，要完成以下 3 步：

- ◎ 程序中某个类声明实现 `Runnable` 接口，并且在这个类中实现 `run()` 方法；
- ◎ 生成这个类的对象；
- ◎ 用 `Thread (Runnable target)` 构造函数生成 `Thread` 对象，其中 `target` 是声明实现了 `Runnable` 接口的对象，并且用 `start()` 方法启动线程。

以上这 3 个步骤用 Java 语言表述，先定义一个实现了 `Runnable` 接口的类：

```
class NewThreadRun implements Runnable
{
    ...
    public void run()
    { ... }
}
```

用下面的代码创建并执行新线程：

```
NewThreadRun n = new NewThreadRun();
Thread thread = new Thread(n);
thread.start();
```

`Runnable` 是 `java.lang` 包中的一个接口。任何一个类都可以实现这个接口，从而实现创建和执行线程的功能。实现 `Runnable` 接口的类必须覆盖接口中定义的 `run()` 方法，它仍然是完成具体任务的地方。

下面举例说明如何用这种方法来编写多线程程序。

【例 12-20】 `RunTest.java` 源文件。

```
public class RunTest
{
    public RunTest()
    {
        FirstThread first = new FirstThread();
        SecondThread second = new SecondThread();
        Thread thread1 = new Thread(first);
        Thread thread2 = new Thread(second);
        thread1.start();
        thread2.start();
    }
    public static void main(String[] args)
    {
```



```
        new RunTest();
    }
}
class FirstThread implements Runnable
{
    public void run()
    {
        try
        {
            System.out.println("First thread starts running.");
            for(int i=0; i<10; i++)
            {
                System.out.println("First " + i);
                Thread.sleep(1000);
            }
            System.out.println("First thread finishes running.");
        }
        catch (InterruptedException e) {}
    }
}
class SecondThread implements Runnable
{
    public void run()
    {
        try
        {
            System.out.println("\tSecond thread starts running.");
            for(int i=0; i<10; i++)
            {
                System.out.println("\tSecond " + i);
                Thread.sleep(1000);
            }
            System.out.println("\tSecond thread finishes running.");
        }
        catch (InterruptedException e) {}
    }
}
```

这个程序与 ThreadTest1.java 有相同的功能，只不过现在用实现 Runnable 接口的方法来

创建和执行线程。在这个程序中应该注意的是，使用 `Thread` 类的所有方法时，必须用对象名或类名加以限制，如程序中的 `thread1.start()`、`Thread.sleep()`。因为不加限制地引用方法，表示默认使用 `this` 引用，即自己或父类定义的方法，但 `FirstThread` 类和 `SecondThread` 类在这个程序中不是 `Thread` 类的子类。

以上两种方法都可以创建和执行线程。前一种方法要求一定是 `Thread` 类的子类。后一种方法可以不是 `Thread` 类的子类，但必须实现 `Runnable` 接口，这种方法使用起来更加灵活。有时只能使用后一种方法，如某类已经定义为 `Applet` 类的子类，由于 Java 不允许多重继承，这时不能再定义它为 `Thread` 类的子类，只有声明其实现 `Runnable` 接口来创建和执行新线程。



12.15 输入/输出流类

计算机不仅对各类信息具有强大的处理能力，而且还能管理各种外部设备，如键盘、串行口、显示器等，它们都是通过 I/O 端口与计算机相连的。计算机要对外设进行有效控制，必须编写相应的外设驱动程序。由于 I/O 设备性质各不相同，且不断更新换代，这增加了计算机管理外设的难度。

为了实现对外设的统一管理，屏蔽不同外设的差异，Java 使用 `java.io` 包来实现上层软件与硬件的隔离，引入流的概念，抽象地把产生数据的源和使用数据的目的联系起来。在实际应用时，把流分为输入流和输出流，输入流连在某个产生数据的设备上，输出流连在某个接收数据的设备上。这样，计算机在处理输入/输出时，只是从输入流中读取数据，把结果写出到输出流中，而不必过问与流相连的具体设备。

流是一种抽象的数据类型，具有长度（元素的个数）、当前位置（任意时刻的唯一存取点）和存取方式（只读、只写或读写）。用户非常熟悉的一种流是传统的磁盘文件，但是流的概念已经扩展为：内存中的流、来自键盘的字符流、送往屏幕的字符流以及出入串行口（或其他设备）的字符流。流的这些概念绝大多数来自 UNIX 系统，即“任何事物都是文件”的思想。



12.15.1 文件系统

操作系统的文件管理是向应用程序提供的最基本服务之一，但在计算机的发展过程中，形成了众多的文件管理系统，它们互不兼容，给用户编程留下了相当大的困难，Java 消除了这种不兼容性。

`File` 类能够处理由本地文件系统维护的具体文件，它提供独立于平台的文件处理方法。`File` 类提供了很多实用方法，下面用实例程序来说明其主要方法的使用。

1. 文件路径和属性

与文件路径有关的方法有：

`getPath()`方法返回 `File` 对象的路径；

`getAbsolutePath()`方法返回 `File` 对象的绝对路径；

`getName()`方法返回 `File` 对象的文件名或目录名。



与 `getPath()` 的含义不同, `getName()` 方法要检查给定字符串中是否含有路径分隔符。如果包含路径分隔符, 则取最后一个分隔符后的字符串作为文件名返回给调用者; 如果不包含路径分隔符, 则 `getName()` 和 `getPath()` 返回相同的字符串。

`getParent()` 返回 `File` 对象的父目录。

另外, `File` 类还有几个方法用来说明文件的属性或状态: `exist()`, `canWrite()`, `canRead()`, `isFile()`, `isDirectory()`, `isAbsolute()`, 它们都返回 `boolean` 型数据, 分别表示: 文件是否存在, 是否写保护, 是否读保护, 是否是文件, 是否是目录, 是否使用绝对路径。

Java 只能查询上述文件属性, 其他属性, 如: 隐藏、系统、归档等属性, 无法查询。`File` 类的平台独立性, 是以放弃某些系统的个性为代价的。

【例 12-21】 下面的程序首先判断给定文件是否存在, 如果存在, 则显示文件路径、绝对路径等, 然后再查询文件的属性。

```
//Program Name: FileTest1.java
import java.io.*;
class FileTest1
{
    public static void main(String[] args)
    {
        String path;
        if(args.length != 1)
        {
            System.err.println("Usage: java FileTest1 File or Dir");
            System.exit(-1);
        }
        File f = new File(args[0]);
        if (f.exists())
        {
            System.out.println("----- ");
            System.out.println("Absolute Path:   " + f.getAbsolutePath());
            System.out.println("File Path:      " + f.getPath());
            System.out.println("File Name:      " + f.getName());
            System.out.println("Parent Directory: " + f.getParent());
            System.out.println("----- ");
            String canRead = f.canRead() ? "Yes" : "No";
            String canWrite = f.canWrite() ? "Yes" : "No";
            String isFile = f.isFile() ? "Yes" : "No";
            String isDir = f.isDirectory() ? "Yes" : "No";
            String isAbs = f.isAbsolute() ? "Yes" : "No";
            System.out.println("Readable:      "+canRead);
```

```

        System.out.println("Writable:      "+canWrite);
        System.out.println("Is directory:   "+isDir);
        System.out.println("Is file:      "+isFile);
        System.out.println("Is absolute path:"+isAbs);
    }
    else System.out.println("Cannot found file: " + args[0]);
}
}

```

这是一个 **Application** 程序，用命令行参数接收输入文件名。读者可以用不同的命令行参数来执行上面的程序，体会上述方法的区别。

2. 创建目录和删除文件

mkdir()和 **makedirs()**用于创建目录。创建目录的位置完全取决于 **File** 对象的路径。

delete()用于删除文件或目录，删除目录时，应该保证所删目录是一个空目录，否则删除操作失败。

【例 12-22】FileTest2.java 源文件。

```

import java.io.*;

class FileTest2
{
    public static void main(String[] args)
    {
        File dir, subdir;
        if(args.length != 1)
        {
            System.err.println("Usage: java FileTest2 <newDirPath>");
            System.exit(-1);
        }
        dir = new File(args[0]);
        if(dir.exists())
            System.out.println(dir.getPath() + " already exist!");
        else
        {
            if(dir.mkdirs())
            {
                System.out.println("Created directory:" + dir.getAbsolutePath());
                subdir = new File(dir, "newSub");
                if (subdir.mkdir())
                {
                    System.out.println("Created subdirectory: " +

```



```
        subdir.getAbsolutePath());
        subdir.delete();
    }
    else
        System.out.println("Could not create subdirectory " +
            subdir.getAbsolutePath());
        dir.delete();
    }
    else
        System.out.println("Could not create directory " +
            dir.getAbsolutePath());
    }
}
```

3. 文件更名

renameTo()方法不但可以给文件更名，还可以给目录更名。

equals()判断两个 File 对象是否相等。可以用它来判断用户给定的原文件名和新文件名是否相等，如果相等，则不能进行更名操作。

【例 12-23】 本例说明如何使用这两个方法。

```
//Program Name: FileRename.java
import java.io.*;
class FileRename
{
    public static void main(String[] args)
    {
        String message;
        if (args.length != 2)
        {
            System.err.println("Usage: java FileRename <file1> <file2>");
            System.exit(-1);
        }
        File f1 = new File(args[0]);
        File f2 = new File(args[1]);
        if (f1.equals(f2))
        {
            System.err.println("Cannot rename a file to itself");
            System.exit(-1);
        }
        message = f1.renameTo(f2) ?
```



```

        " renamed to " : " could not be renamed to ";
    System.out.println(f1.getPath() + message + f2.getPath());
    }
}

```

4. 目录清单

list()方法产生目录清单,它只返回指定目录中包含的文件名或子目录名,没有文件长度、修改时间、文件属性等信息。

lastModified()返回文件最后一次被修改的时间,其值是相对于 1970 年 1 月 1 日的时间毫秒数,为了便于阅读,必须将其变成 java.util.Date 对象。

【例 12-24】FileDir.java 源文件。

```

import java.io.File;
import java.util.Date;
class FileDir
{
    public static void main(String[] args)
    {
        if (args.length != 1)
        {
            System.err.println("Usage: java FileDir filepath");
            System.exit(-1);
        }
        File f = new File(args[0]);
        String[] ls = f.list();
        for (int i = 0; ls != null && i < ls.length; i++)
            printOne(new File(f, ls[i]));
    }
    public static void printOne(File f)
    {
        if (f.exists())
        {
            System.out.print(f.canRead() ? "r" : "-");
            System.out.print(f.canWrite() ? "w" : "-");
            System.out.print(f.isDirectory() ? "d" : "-");
            if(!f.isDirectory())
                System.out.print("\t\t" + f.length());
            else
                System.out.print("\t<Dir>\t");
            System.out.print("\t");
        }
    }
}

```



```
        System.out.print(new Date(f.lastModified()));
        System.out.print("\t");
    }
    else
    {
        System.out.print("\t\t\t\t\t");
    }
    System.out.println(f.getName());
}
}
```

说明：File 类不允许访问文件内容，即读/写文件，也不能改变文件的属性。

12.15.2 读/写文件

File 类具有查询文件属性、状态和文件名等功能，但不能访问文件内容。在 java.io 包中，RandomAccessFile 类和输入/输出流类具有读/写文件的功能。RandomAccessFile 类只能进行文件的输入/输出，而流类功能更加强大，它可以进行文件的一切输入/输出操作。

RandomAccessFile 类的定义如下：

```
public class RandomAccessFile implements DataOutput, DataInput
{
    public RandomAccessFile(String name, String mode)
        throws IOException;
    public RandomAccessFile(File file, String mode) throws IOException;
    //misc methods
    public final FileDescriptor getFD() throws IOException;
    public long getFilePointer() throws IOException;
    public void seek(long pos) throws IOException;
    public long length() throws IOException;
    public void close() throws IOException;
    public int read() throws IOException;
    public int read(byte b[], int off, int len) throws IOException;
    public int read(byte b[]) throws IOException;
    //DataInput interface
    public final void readFully(byte b[]) throws IOException;
    public final void readFully(byte b[], int off, int len)
        throws IOException;
    public int skipBytes(int n) throws IOException;
    public final boolean readBoolean() throws IOException;
```

```

public final byte readByte() throws IOException;
public final int readUnsignedByte() throws IOException;
public final short readShort() throws IOException;
public final int readUnsignedShort() throws IOException;
public final char readChar() throws IOException;
public final int readInt() throws IOException;
public final long readLong() throws IOException;
public final float readFloat() throws IOException;
public final double readDouble() throws IOException;
public final String readLine() throws IOException;
public final String readUTF() throws IOException;
//DataOutput interface
public void write(int b) throws IOException;
public void write(byte b[]) throws IOException;
public void write(byte b[], int off, int len) throws IOException;
public final void writeBoolean(boolean v) throws IOException;
public final void writeByte(int v) throws IOException;
public final void writeShort(int v) throws IOException;
public final void writeChar(int v) throws IOException;
public final void writeInt(int v) throws IOException;
public final void writeLong(long v) throws IOException;
public final void writeFloat(float v) throws IOException;
public final void writeDouble(double v) throws IOException;
public final void writeBytes(String s) throws IOException;
public final void writeChars(String s) throws IOException;
public final void writeUTF(String str) throws IOException;
}

```

它提供了两个构造函数。参数的含义说明如下。

name: 是一个 **String** 对象，表示被访问的文件名。

file: 是一个 **File** 对象，表示被访问的文件名。用这种方式提供文件名，应用程序独立于平台。

mode: 用字符串表示被访问文件的读/写模式，"r"表示文件以只读方式打开，"rw"表示文件以读/写方式打开。

以读/写方式生成 **RandomAccessFile** 对象时，如果该文件不存在，则创建该文件，供程序进行读/写操作；如果该文件已经存在，则以覆盖方式（不是改写方式）把输出数据写入到文件中，原文件中没有被覆盖的部分，仍然保留在文件之中。在 **RandomAccessFile** 类中没有专门打开文件的方法，在生成 **RandomAccessFile** 对象时，文件即被打开，可供程序访问，这时文件读/写指针为 0，即读/写位于文件的开头。



`RandomAccessFile` 类实现了 `DataInput` 和 `DataOutput` 两个接口，这两个接口分别定义了读入和写出的方法。`DataInput` 和 `DataOutput` 中定义的方法基本上是相对应的，即在 `DataInput` 中定义了一个读入数据的方法，则在 `DataOutput` 中就有一个结构相似的写出方法。但 `DataInput` 接口多出几个与写出操作无关的方法。

`skipBytes()`: 在输入流中跳过 n 个字节

`readUnsignedByte()`: 读入一个无符号字节数据。

`readLine()`: 读入一行字符。有 4 种行结束符：回车符 `\r`，换行符 `\n`，回车符后紧跟换行符，或者文件结束。读入的一行字符中包含行结束符。

由于读入和写出方法的含义相似，因此我们用写出方法来说明不同格式的含义。有的写出方法带有不同的后缀，分别为 `Byte`、`Short`、`Int`、`Long`、`Float`、`Double`、`Char`、`Boolean`，这些方法表示写出 Java 基本数据类型的数据，例如，`writeInt()` 表示写出一个 `int` 型整数。对于用这些方法写出的数据，应该用带有相同后缀的读入方法来读取数据，否则可能出错。这些方法中 `writeBoolean()` 比较特殊，写出数据时，用 0 表示 `false`，用 1 表示 `true`。而用 `readBoolean()` 读入数据时，如果是 0，则返回 `false`，非 0 则返回 `true`。

有 3 个方法与字符串的写出有关。

`writeBytes()`: 写出字符串时，每个字符占一个字节，即 8 位。

`writeChars()`: 在支持 Unicode 码的机器上写出字符串，每个字符占 2 个字节，即 16 位。

`writeUTF()`: 以 8 位编码的方式写出字符串。其中的前 2 个字节，表示以后将要写出数据的字节总数，其值由系统统计。从第 3 个字节开始写出字符串数据，每个字符占 1 个字节。例如，`writeUTF("abc")`，则其结果用十六进制数表示为：00H, 03H, 61H, 62H, 63H，共写出 5 个字节，前 2 个字节的值表示其后还将写出 3 个字节。

另外有 3 个不带任何后缀的写出方法，按字节写出数据。

`write(int b);`

`write(byte b[]);`

`write(byte b[], int off, int len);`

其参数的含义如下。

b: 1 个字节数据或字节数组。

off: 表示从数组中第几个字节开始写出数据。

len: 表示写出字节的总数。

【例 12-25】 下面的程序说明向文件中写出数据的方法。

```
//Program Name: RandomTest.java
import java.io.*;
class RandomTest
{
    public static void main(String[] args)
    {
        RandomAccessFile raf = null;
        if (args.length != 1)
```

```
{
    System.err.println("Usage: java RandomTest <output file>");
    System.exit(-1);
}
try
{
    raf = new RandomAccessFile(args[0], "rw");
    char a = 'a';
    byte b = 2;
    String c = "abc";
    byte[] b2 = {'a', 'b', 'c'};
    raf.write(b);
    raf.write(b2, 0, b2.length);
    raf.writeBoolean(true);
    raf.writeChar(a);
    raf.writeBytes(c);
    raf.writeChars(c);
    raf.writeUTF(c);
    raf.writeUTF("abc\n");
    System.out.println("Length of file: " + raf.length());
}
catch (IOException e)
{
    System.err.println(e);
}
finally
{
    try
    {
        raf.close();
    }
    catch (Exception e)
    {
        System.err.println(e);
    }
}
}
```



程序执行后，文件包含的内容用十六进制数表示为：

02 61 62 63 01 00 61 61 62 63 00 61 00 62 00 63 00 03 61 62 63 00 04 61 62 63 0a

`RandomAccessFile` 类除具有读/写文件的方法外，还定义了与文件操作相关的一些方法。

`seek()`：支持随机访问文件，它通过移动文件的读/写指针实现文件的随机读/写。其参数表示读/写指针相对于文件起始位置的偏移量。

`getFilePoint()`：返回当前文件指针的位置。

`close()`：关闭文件和释放与打开文件有关的资源。在使用完文件或使用中出现异常后，都应该关闭文件。在处理异常的 `try-catch-finally` 结构中，无论是 `try` 内程序段正常结束，还是发生异常后，都要执行 `finally` 中的程序段，所以，`close()` 语句可以安排在 `finally` 中。

【例 12-26】 下面的程序把一个文件复制成另一个文件。

```
//Program Name: RandCopy.java
import java.io.*;
class RandCopy
{
    public static void main(String args[])
    {
        RandomAccessFile raf1 = null, raf2 = null;
        long fileSize = -1;
        byte[] buffer;
        if(args.length != 2)
        {
            System.out.println("Usage: java RandCopy <file1><file2>");
            System.exit(0);
        }
        if(args[0].equals(args[1]))
        {
            System.out.println(args[0]);
            System.out.println("File cannot copied onto itself!");
            System.exit(-1);
        }
        try
        {
            raf1 = new RandomAccessFile(new File(args[0]), "r");
            //其中 raf1 表示原文件，以只读方式打开
            fileSize = raf1.length();//获得原文件的长度
        }
        catch (IOException e)
        {
            //异常处理
        }
    }
}
```

```
        System.out.println("Cannot find " + args[0]);
        System.exit(-1);
    }
    try
    {
        raf2 = new RandomAccessFile(new File(args[1]), "rw");
        // raf2 表示新复制文件，以读/写方式打开
    }
    catch (IOException e)
    {
        System.out.println("Cannot open " + args[1]);
        System.exit(-1);
    }
    buffer = new byte[(int)fileSize];
    try
    {
        raf1.readFully(buffer, 0, (int) fileSize);
        //一次把所有数据读入内存中
        raf2.write(buffer, 0, (int) fileSize);
        //全部写出到新文件中
    }
    catch(IOException e)
    {
        System.out.println("Cannot copy file " + args[0] + "to " + args[1]);
    }
    finally
    {
        try
        {
            raf1.close();
            raf2.close();
        }
        catch(Exception e)
        {
            System.err.println(e);
        }
    }
}
```



12.15.3 抽象流类

流具有处理输入/输出的功能，java.io 包中的类以流类为主。Java 语言定义了两个抽象类：InputStream 类和 OutputStream 类，还派生了很多与流有关的子类。

1. InputStream 类

InputStream 类的定义如下：

```
public abstract class InputStream extends Object
{
    public InputStream();
    public abstract int read() throws IOException;
    public int read(byte b[]) throws IOException ;
    public int read(byte b[], int off, int len) throws IOException ;
    public long skip(long n) throws IOException ;
    public int available() throws IOException ;
    public void close() throws IOException ;
    public synchronized void mark(int readlimit) ;
    public synchronized void reset() throws IOException ;
    public boolean markSupported() ;
}
```

InputStream 类提供了有关读入数据的方法，它定义的读入方法，读入数据时都是以字节为单位的，可以一个字节一个字节地读入数据，也可以读入任意长度的字节块。

类中 mark()、reset()、markSupported()三个方法与标记复位操作有关，它们的作用是：mark()在输入流（如文件）中任意位置做一标记，从标记位置开始，以后读入的字节数据可以用 reset()方法取消，使输入流的读入位置复位到标记处。

完成这种标记复位功能的前提条件是：markSupported()方法返回值必须为 true。如果返回值为 false，则表示不支持这种功能。InputStream 类中的 markSupported()返回值为 false，若希望使用这种功能的 InputStream 子类，则必须覆盖 markSupported()方法并且使返回值为 true。

在使用这种标记复位功能时，mark()所需 readlimit 参数代表一个极限值，它表示从标记位置开始，到用 reset()取消读入数据时，最多能读入的字节数。如果读入的字节数已经超过了 readlimit 限制，再使用 reset()，则复位的位置不确定。如果 readlimit 设置为 100 个字节，而在使用 reset()时，相对于标记位置，已经读入了 150 个字节，那么，复位后的位置不能保证在标记位置，此时，标记功能已失去意义。所以，在编程中使用标记复位功能时，应该小心谨慎。

所有 InputStream 的子类都是针对不同的输入数据源的，其类名的前缀清楚地表示出输入的数据源，如 FileInputStream 类的数据源是文件，PipedInputStream 类的数据源是管道等。FilterInputStream 类及其子类是加强流，它的功能更加强大，使用更加灵活。

2. OutputStream类

与输入流类相对应的另一个抽象类是 `OutputStream` 类，它定义了与输出操作有关的方法，其定义如下：

```
public abstract class OutputStream extends Object
{
    public OutputStream();
    public abstract void write(int b) throws IOException;
    public void write(byte b[]) throws IOException;
    public void write(byte b[], int off, int len) throws IOException;
    public void flush() throws IOException;
    public void close() throws IOException;
}
```

输出流写出数据也是以字节为单位的，即可以一个字节一个字节地写出数据，也可以一次写出任意长度的字节块。

`OutputStream` 类的子类：`ByteArrayOutputStream` 类、`FileOutputStream` 类和 `PipedOutputStream` 类表示写出不同的数据类型，其前缀表示了不同的数据类型。另一个子类 `FilterOutputStream` 类和它的子类加强输出流的功能。

12.15.4 文件输入/输出流类

`FileInputStream` 类和 `FileOutputStream` 类分别完成对文件的输入/输出（即读/写）操作。

1. FileInputStream类

`FileInputStream` 类提供从文件中读入数据的方法，其定义为：

```
public class FileInputStream extends InputStream
{
    public FileInputStream(String name) throws FileNotFoundException;
    public FileInputStream(File file) throws FileNotFoundException;
    public FileInputStream(FileDescriptor fdObj);
    public int read() throws IOException;
    public int read(byte b[]) throws IOException;
    public int read(byte b[], int off, int len) throws IOException;
    public long skip(long n) throws IOException;
    //使文件的读入指针向前或向后移动 n 个字节
    //其参数值相对于当前文件指针
    public int available() throws IOException;
    //返回文件输入流中可供读入的字节数，实际上就是文件的长度
    public void close() throws IOException;
```



```
public final FileDescriptor getFD() throws IOException;
protected void finalize() throws IOException;
}
```

它有三个构造函数，各自使用不同的参数。

name: 用 `String` 对象表示文件名。

file: 用 `File` 对象表示文件名，使应用程序独立于平台。

fdObj: 用 `FileDescriptor` 对象表示文件名，`FileDescriptor` 类定义一个本地文件系统。

在生成 `FileInputStream` 对象时，同时打开文件以供应用程序读入数据。`FileInputStream` 类主要覆盖了 `InputStream` 类的读入数据的方法，有三个 `read()` 方法，它们都按字节读入数据。不带参数的 `read()` 一次只能读入一个字节，另外两个 `read()` 一次可以读入任意多个字节。在读入数据后，前一个 `read()` 方法返回读入的字节数据，后两个方法返回读入的字节数。如果遇到文件结束符，则返回-1。

【例 12-27】 下面的程序说明 `FileInputStream` 类的使用方法。程序实现类似 MS-DOS 操作系统中 `type` 命令，用于显示文件内容。

```
//Program Name: FileType.java
import java.io.*;
class FileType
{
    public static void main(String[] args)
    {
        if (args.length != 1)
        {
            System.err.println("Usage: java FileType <input_file>");
            System.exit(-1);
        }
        File file = new File(args[0]);
        try
        {
            FileInputStream in = new FileInputStream(file);
            int c;
            int i = 0;
            while ((c = in.read()) > -1)
            {
                if ((char)c == '\n') i++;
                System.out.print((char)c);
            }
            in.close();
            System.out.flush();
        }
    }
}
```

```

        System.out.println("\n\n-----");
        System.out.println("File " + args[0] + " Lines: " + i);
    }
    catch (FileNotFoundException e)
    {
        System.err.println(file + " is not found");
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
}
}

```

程序在生成 `FileInputStream` 对象时，用 `File` 对象给出文件名，虽然比用字符串给出文件名稍麻烦些，但这种方法使应用程序独立于平台，是一种良好的设计习惯。

程序在显示文件内容时使用了 `System.out.print()` 方法，而不是以前程序中常用的方法 `System.out.println()`。两者的区别是：`println()` 执行时，立即把字符显示在屏幕上，并且每次调用 `println()` 方法，它总是在新的一行输出数据。`print()` 执行时，并不立即显示数据，而把数据缓存在输出流中，直到遇到新行字符 `\n` 或执行了 `flush()` 方法后，才使字符显示在屏幕上。如果程序中使用 `println()`，那么在屏幕上的显示情况就是一行一个字符，不是原来文件的行格式。

2. `FileOutputStream` 类

`FileOutputStream` 类提供把数据写出到文件中的方法，其定义为：

```

public class FileOutputStream extends OutputStream
{
    public FileOutputStream(String name) throws IOException;
    public FileOutputStream(String name, boolean append)
        throws IOException;
    public FileOutputStream(File file) throws IOException;
    public FileOutputStream(FileDescriptor fdObj);
    public void write(int b) throws IOException;
    public void write(byte b[]) throws IOException;
    public void write(byte b[], int off, int len) throws IOException;
    public void close() throws IOException;
    public final FileDescriptor getFD() throws IOException;
}

```

`FileOutputStream` 类和 `FileInputStream` 类在构造函数和方法的参数定义上十分相似，不同之处是，一个针对输入，另一个针对输出。



生成 `FileOutputStream` 对象时, 如果文件不存在, 则创建该文件供程序输出数据。如果文件已经存在, 则有改写和附加两种输出数据的方式:

- ◎ 改写的含义是, 先把原文件长度截为零, 原文件数据被丢弃, 然后再输出数据;
- ◎ 附加的含义是在原文件末尾追加输出数据, 原文件数据仍然存在。

除使用第 2 个构造函数生成对象外, 其余都是以改写方式输出数据。第 2 个构造函数中增加了 `append` 参数, 可以选择输出数据的方式, 该参数为 `true`, 表示附加方式, 为 `false`, 表示改写方式。

说明: `FileInputStream` 类和 `FileOutputStream` 类具有读/写文件的方法, 但它们读/写数据的效率并不高, Java 鼓励程序员用更有效的方法覆盖有关读/写方法, 以提高应用程序的性能。

12.15.5 加强输入/输出流类

我们把 `FilterInputStream` 类和 `FilterOutputStream` 类以及它们的子类, 称为加强输入/输出流类, 因为, 利用这些流类不但可以改进程序的输入/输出性能, 而且功能更加强大。

`FilterInputStream` 类的子类包括:

- ◎ `BufferedInputStream` 类
- ◎ `DataInputStream` 类

`FilterOutputStream` 类的子类包括:

- ◎ `BufferedOutputStream` 类
- ◎ `DataOutputStream` 类

1. `FilterInputStream`类和`FilterOutputStream`类

`FilterInputStream` 类和 `FilterOutputStream` 类与 `InputStream` 类和 `OutputStream` 类相比, 并没有增加任何新的方法, 只是分别覆盖 `InputStream` 类和 `OutputStream` 类的所有方法。它们真正的改变是在其构造函数上:

```
protected FilterInputStream(InputStream in)
protected FilterOutputStream(OutputStream out)
```

它们要求的参数分别是一个 `InputStream` 对象和一个 `OutputStream` 对象, 即任意一个输入流或输出流对象, 而不是 `InputStream` 类和 `OutputStream` 类要求的 `String` 或 `File` 对象。

说明: 在 `FilterInputStream` 类中虽然覆盖了有关标记复位的方法, 但它仅仅是调用 `InputStream` 类的相应方法, 所以, `FilterInputStream` 类并不支持标记复位功能。

2. `BufferedInputStream`类和`BufferedOutputStream`类

这两个类在标准读/写功能和应用程序之间, 增加输入/输出缓冲机制, 从而显著地提高输入/输出的速度。

`BufferedInputStream` 类用于提高读入数据的速度, 构造函数如下:

```
public BufferedInputStream(InputStream in);
public BufferedInputStream(InputStream in, int size);
```

它的构造函数使用输入流对象,在生成 `BufferedInputStream` 对象时可以设置输入缓冲区的大小,也可以使用默认值,其值是 2048 字节。生成 `BufferedInputStream` 对象后,应用程序在读入数据时,直接取自缓存区,从而提高读入数据的速度。

`BufferedOutputStream` 类用于提高写出数据的速度,它与 `BufferedInputStream` 类相似,使用输出缓冲区来提高输出数据的速度,输出缓冲区的默认大小为 512 字节。

3. `DataInputStream`类和`DataOutputStream`类

结合 `RandomAccessFile` 类,就很容易理解这两个类,因为, `RandomAccessFile` 类同时实现了两个接口: `DataInput` 接口和 `DataOutput` 接口,而 `DataInputStream` 类和 `DataOutputStream` 类分别实现了这两个接口。所以, `DataInputStream` 类中读数据的方法和 `DataOutputStream` 类中写数据的方法,与 `RandomAccessFile` 类中的读/写方法的含义完全一致,在此不再赘述。

12.15.6 其他输入/输出流类

本节将介绍的输入流类有:

`SequenceInputStream` 类

`PipedInputStream` 类

输出流类有:

`PipedOutputStream` 类

1. `SequenceInputStream`类

`SequenceInputStream` 类可以将两个或几个输入流不露痕迹地接合在一起,生成一个长长的接合流。在读入数据时,它忽略前面几个输入流的结束符 EOF,直到最后一个流的结束符 EOF 时,才完成流的输入,其定义如下:

```
public class SequenceInputStream extends InputStream
{
    public SequenceInputStream(Enumeration e);
    //能够接合任意多个输入流
    public SequenceInputStream(InputStream s1, InputStream s2);
    //只能接合两个输入流
    public int read() throws IOException;
    public int read(byte buf[], int pos, int len) throws IOException;
    public int available() throws IOException;
    public void close() throws IOException;
}
```



【例 12-28】下面程序把两个文件输入流接合成一个 `SequenceInputStream` 流，再写到一个文件中。

```
//Program Name: SeqTest.java
import java.io.*;
class SeqTest
{
    public static void main(String[] args)
    {
        FileInputStream fis1 = null;
        FileInputStream fis2 = null;
        FileOutputStream fos = null;
        SequenceInputStream sis = null;
        int ch;
        if(args.length != 3)
        {
            System.out.println("Usage: java SeqTest <file1><file2><outfile>");
            System.exit(0);
        }
        try
        {
            fis1 = new FileInputStream(new File(args[0]));
            fis2 = new FileInputStream(new File(args[1]));
            fos = new FileOutputStream(new File(args[2]));
        }
        catch (IOException e)
        {
            System.out.println("Cannot find file!");
            System.exit(-1);
        }
        sis = new SequenceInputStream(fis1, fis2);
        try
        {
            while((ch = sis.read()) > -1) fos.write(ch);
            fis1.close();
            fis2.close();
            sis.close();
        }
        catch (IOException e)
```

```

{
    System.out.println(e);
    System.exit(-1);
}
}
}

```

2. 管道输入/输出流类

管道是 UNIX 的发明，它大大增强了流的概念。其含义如下：

```
c:\>dir|sort|more
```

这里使用了两个管道（用|表示）把三个命令连在一起，第一个管道把 dir 命令的输出连接到 sort 命令的输入，第二个管道将 sort 命令的输出连到 more 命令的输入，从而实现生成目录清单，进行排序后，一页一页地显示出来。

Java 使用了管道技术，用 PipedInputStream 类和 PipedOutputStream 类实现管道操作，这两个类必须同时使用，它们的定义分别为：

```

public class PipedInputStream extends InputStream
{
    public PipedInputStream(PipedOutputStream src) throws IOException;
    public PipedInputStream();
    public synchronized int read() throws IOException;
    public synchronized int read(byte b[], int off, int len)
        throws IOException;
    public synchronized int available() throws IOException;
    public void close() throws IOException;
    public void connect(PipedOutputStream src) throws IOException;
}

public class PipedOutputStream extends OutputStream
{
    public PipedOutputStream(PipedInputStream snk)
        throws IOException;
    public PipedOutputStream();
    public void write(int b) throws IOException;
    public void write(byte b[], int off, int len) throws IOException;
    public synchronized void flush() throws IOException;
    public void close() throws IOException;
    public void connect(PipedInputStream snk) throws IOException;
}

```

管道技术在多线程系统中使用较多，它可以解决不同线程之间同步通信的问题，从而保证大量数据交换顺利进行。要实现多线程的同步通信，必须把管道连接起来。这里有两种方



法实现管道的连接：

- ◎ 使用带参数的构造函数；
- ◎ 使用不带参数的构造函数，再调用任何一个类的 `connect()`方法，实现输入管道和输出管道的连接。

【例 12-29】下面的程序使用两个线程：一个线程模拟数据采集，用 `Random` 类随机生成数值；另一个线程模拟数据处理，使用数据，计算其平均值。

```
//Program Name: PipeTest.java
import java.io.*;
import java.util.Random;
class RunningAverage extends Thread
{
    private DataInputStream in;
    double total = 0;
    long count = 0;
    public RunningAverage(InputStream i)
    {
        in = new DataInputStream(i);
    }
    public void run()
    {
        while (true)
        {
            try
            {
                double num = in.readDouble();
                total += num;
                count++;
                System.out.println(count+":"+num+"\t avg="+total/count);
            }
            catch (IOException e)
            {
                e.printStackTrace();
            }
        }
    }
}
class NumberGenerator extends Thread
{

```



```
private DataOutputStream out;
private Random gen = new Random();
private final long RANGE = 1000;
public NumberGenerator(OutputStream o)
{
    out = new DataOutputStream(o);
}
public void run()
{
    while (true)
    {
        try
        {
            double num = gen.nextDouble() * RANGE;
            out.writeDouble(num);
            out.flush();
            sleep(500);
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }
}
class PipeTest
{
    public static void main(String[] args)
    {
        try
        {
            PipedOutputStream producer = new PipedOutputStream();
            PipedInputStream consumer = new PipedInputStream(producer);
            RunningAverage avg = new RunningAverage(consumer);
```



```
NumberGenerator gen = new NumberGenerator(producer);
gen.start();
avg.start();
try
{
    Thread.sleep(5000);
}
catch (InterruptedException e) {}
gen.stop();
avg.stop();
producer.close();
consumer.close();
}
catch (IOException e)
{
    e.printStackTrace();
}
}
```

上面程序用 `PipedOutputStream()` 生成管道输出流 `producer`，用 `PipedInputStream(producer)` 生成管道输入流 `consumer`，同时实现了两个管道的连接。程序中的两个类 `RunningAverage` 和 `NumberGenerator`，都是 `Thread` 类的子类，是两个不同的线程。前一个类生成数据，用管道输出这些数据；后一个类从输入管道接收数据，然后计算平均值。`main()` 在启动了这两个线程工作后，睡眠 5 秒，然后用 `stop()` 停止两个线程，再用 `close()` 关闭管道输入/输出流，结束程序。

3. 对象流和序列化

`ObjectInputStream` 类和 `ObjectOutputStream` 类分别是 `InputStream` 和 `OutputStream` 的子类，这两个类所创建的对象分别称为对象输入流和对象输出流。对象输出流使用 `writeObject(Object obj)` 方法将一个对象 `obj` 写到输出流中，送往目的地。对象输入流使用 `readObject()` 从源读取一个对象到程序中。

构造方法分别是：

```
ObjectInputStream(InputStream in)
```

```
ObjectOutputStream(OutputStream out)
```

当使用对象流写出或读入对象时，要保证对象是序列化的。一个类如果实现了 `Serializable` 接口，那么这个类的对象就是序列化的对象。`Serializable` 接口中的方法对程序是不可见的，因此实现该接口的类不需要实现额外的方法。当把一个序列化对象写出或读入到对象流中的时候，JVM 会实现 `Serializable` 接口中的方法。

【例 12-30】TestObj.java 源文件。

```

import java.io.*;

class Student implements Serializable //实现 Serializable 接口
{
    String name=null;
    double height;
    Student(String name,double height)
    {
        this.name=name;
        this.height=height;
    }
    public void setHeight (double c)
    {
        this.height=c;
    }
}

public class TestObj
{
    public static void main(String args[])
    {
        Student zhang=new Student("张三",1.65);
        try
        {
            //将对象 zhang 写入到文件 s.txt 中
            FileOutputStream file_out=new FileOutputStream("s.txt");
            ObjectOutputStream object_out=new ObjectOutputStream(file_out);
            object_out.writeObject(zhang);
            //从文件 s.txt 中读对象
            FileInputStream file_in=new FileInputStream("s.txt");
            ObjectInputStream object_in=new ObjectInputStream(file_in);
            Student li=(Student)object_in.readObject();
            li.setHeight(1.78); //设置身高
            li.name="李四";
            System.out.println(zhang.name+" 身高是: "+zhang.height);
            System.out.println(li.name+" 身高是: "+li.height);
        }
        catch(ClassNotFoundException event)
        {
            System.out.println("不能读出对象");
        }
    }
}

```



```
    }  
    catch(IOException event)  
    {  
        System.out.println("can not read file"+event);  
    }  
}  
}
```

运行结果为：

张三身高是：1.65

李四身高是：1.78



12.15.7 Reader和Writer

`InputStream` 类和 `OutputStream` 类及其子类，在读/写流内数据时以字节为单位。`Reader` 类和 `Writer` 类及其子类，在读/写流内数据时以字符为单位。常用的输入/输出流类如图 12-8 所示。

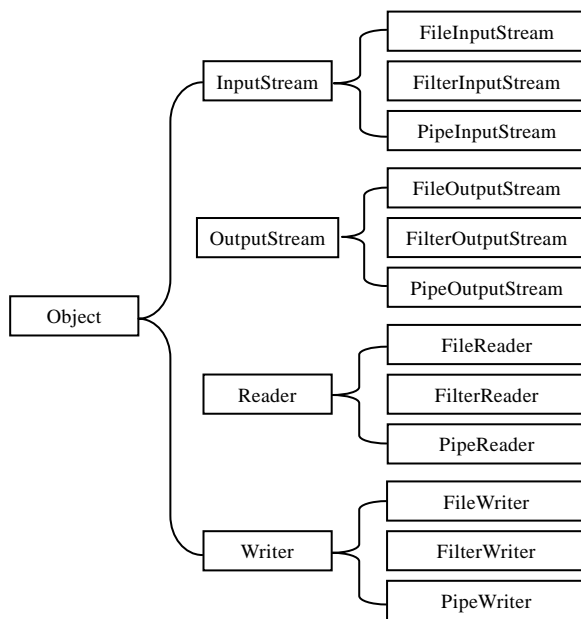


图 12-8 输入/输出流类

两种流类中定义的方法的种类和参数大致相同。

例如：

```
public int read(byte b[], int off, int len); //InputStream 类的方法
```

```
public int read(char b[], int off, int len); //Reader 类的方法
```



12.16 网络编程

IP 地址标识 Internet 上的计算机，端口号标识正在计算机上运行的进程（程序），端口号与 IP 地址的组合得出一个网络套接字。端口号的范围为： $0 \sim 2^{16}$ 。端口号一般分为两类。

- ◎ 知名端口（Well-Known Ports）：范围为 $0 \sim 1023$ 。这些端口号一般固定分配给一些服务。
- ◎ 动态端口（Dynamic Ports）：范围为 $1024 \sim 65535$ 。当程序向系统提出访问网络的申请时，系统就从这些端口号中分配一个未用的供该程序使用。在关闭程序进程后，就会释放所占用的端口号。

java.net 包中包含与网络编程相关的一些基本类，按功能不同可分为：

- ◎ Internet 寻址——InetAddress 类和 URL 类；
- ◎ UDP/IP 无连接服务类——DatagramPacket 类和 DatagramSocket 类；
- ◎ TCP/IP——Socket 类和 ServerSocket 类；
- ◎ WWW——URL 类和 URLConnection 类。



12.16.1 InetAddress 类

InetAddress 类提供有关从域名地址查询 IP 地址的方法，其定义如下：

```
public final class InetAddress implements Serializable
{
    public static InetAddress getByName(String host)
        throws UnknownHostException;

    public static InetAddress[] getAllByName(String host)
        throws UnknownHostException;

    public static InetAddress getLocalHost(String host)
        throws UnknownHostException;

    public String getHostName();//获取主机名
    public byte[] getAddress();
    //获取整数 IP 地址，需转换才可正确显示
    public String.getHostAddress();//获取 IP 地址
    public boolean equals(Object obj);
    public int hashCode();
    public boolean isMulticastAddress();
    public String toString();//获取主机名/IP 地址
}
```



【例 12-31】TestAdd.java 源文件。

```
import java.net.*;
import java.io.*;
public class TestAdd
{
    public static void main(String args[])
    {
        try
        {
            if(args.length==1)
            {
                InetAddress ipa=InetAddress.getByName(args[0]);
                System.out.println("Host name:"+ipa.getHostName());
                //获取主机名
                System.out.println("Host Address:"+ipa.getHostAddress());
                //获取 IP 地址
                System.out.println("Host IP Address:"+ipa.toString());
                //获取主机名/IP 地址
                System.out.println("Localost:"+InetAddress.getLocalHost());
                String s=ipa.isMulticastAddress()? " Yes" : " No";
                System.out.println("is MulticastAddress:"+s);
                byte[] addr=ipa.getAddress();
                System.out.println("Use getAddress():
                    "+addr[0]+"."+addr[1]+"."+addr[2]+"."+addr[3]);
                System.out.println("Converted getAddress():
                    "+(addr[0]&0XFF)+"."+((addr[1]&0XFF))+"."
                    +(addr[2]&0XFF)+"."+((addr[3]&0XFF)));
            }
        }
        else
        {
            System.out.println("Please enter a hostname:");
        }
    }
    catch(UnknownHostException e)
    {
        {
            e.printStackTrace();
        }
    }
}
```

在某台联网机器上的运行结果为：

```
D:\wjjava\net>java TestAdd www.uestc.edu.cn
Host name:www.uestc.edu.cn
Host Address:202.112.14.173
Host IP Address:www.uestc.edu.cn/202.112.14.173
Local Host:Jane/210.41.107.205
is MulticastAddress: No
Use getAddress():-54.112.14.-83
Converted getAddress():202.112.14.173
```

12.16.2 Socket类和ServerSocket类

Socket API 是 1981 年在 BSD4.1 UNIX 系统中提出的。通过 Socket API，网络应用程序明确地创建、使用及释放套接字，它采用了 Client/Server 编程模式。Socket API 提供以下两类传输服务：

- ◎ 不可靠的数据报传输；
- ◎ 可靠的字节流传输。

套接字可以理解成是由一个主机本地应用程序创建的，为操作系统所控制的接口（门），应用进程通过这个接口跨网络发送/接收消息到/从其他应用进程，因此 Socket 就是应用进程和传输层协议（UCP 或 TCP）之间的门。

对于基于 TCP 的 Socket 编程，关键是如何进行从一个进程到另一个进程的字节流的可靠传输。

- ① 服务器进程必须先运行，服务器进程必须创建套接字（门）来迎候客户的初始联系。
- ② 客户必须初始联系服务器，创建客户本地 TCP Socket，指定服务器进程的 IP 地址、端口号，一旦客户创建套接字，客户 TCP 就发起 3 次握手并建立与服务器 TCP 的连接。
- ③ 一旦客户初始联系（敲门）服务器，服务器 TCP 就为服务器进程创建一个新的 Socket 与客户进程通信。

允许服务器与多个客户通信，可以通过源端口号来区分客户。从应用程序的角度来看，TCP 为客户和服务器提供了可靠的、顺序的、字节流的传输“管道”。

1. 通信机制

Server 端：

- ◎ 创建 ServerSocket 对象，在某个端口提供监听服务；
- ◎ 等待来自 Client 端的请求服务；
- ◎ 接收 Client 端的请求，用返回的 Socket 建立连接；
- ◎ 通过向 Socket 读/写数据实现与 Client 端的通信；
- ◎ 关闭 Socket，结束当前通信，等待其他请求；
- ◎ 关闭 ServerSocket 对象，结束监听服务。

Client 端：

- ◎ 创建 Socket 对象，向 Server 的监听端口发出请求；
- ◎ 通过向 Socket 读/写数据实现与 Server 端的通信；
- ◎ 关闭 Socket，结束与 Socket 通信。



2. Socket类

构造函数:

```
public Socket(String host, int port);
public Socket(InetAddress address, int port) throws IOException;
public Socket(String host, int port, InetAddress localAddr, int localPort)
    throws IOException;
public Socket(InetAddress address, int port, InetAddress localAddr,
    int localPort) throws IOException;
//主机, 端口号, 客户地址, 客户端口号
```

常用方法:

```
public InputStream getInputStream() throws IOException;
//获得从服务器到客户机的数据输入流
public OutputStream getOutputStream() throws IOException;
//获得从客户机到服务器的数据输出流
public synchronized void close() throws IOException;
```

3. ServerSocket类

构造函数:

```
public ServerSocket(int port) throws IOException;
public ServerSocket(int port, int backlog) throws IOException;
public ServerSocket(int port, int backlog, InetAddress bindAddr)
    throws IOException;
```

port: 服务器守候的端口号。

backlog: 允许同时连入服务器的客户机数目, 默认值为 50。

bindAddr: 指该端口捆绑的 IP 地址, 常用于多地址的主机。

创建一个 ServerSocket 类:

```
ServerSocket myServer = new ServerSocket(port);
```

监听可能的 Client 请求:

```
Socket linkSocket = myServer.accept();
```

监听端口, 使 Server 端口的程序处于等待状态。

程序将一直堵塞, 直到捕捉到 Client 端的请求, 并返回一个与该 Client 通信的 Socket 对象为止; 以后 Server 程序只要向这个 Socket 对象读/写数据, 就可向远端的 Client 读/写数据。

结束监听:

```
myServer.close();
```

【例 12-32】基于 TCP 的套接字编程包含: 一个服务器端程序和一个客户端程序。

```
//客户端程序 ClientTest.java
import java.io.*;
```



```

import java.net.*;
public class ClientTest
{
    public static void main(String[] args)
    {
        try
        {
            String s;
            String line="";
            Socket mySocket = new Socket(InetAddress.getLocalHost(),800);
            PrintStream os = new PrintStream(
                new BufferedOutputStream(mySocket.getOutputStream()));
            DataInputStream dis =
                new DataInputStream(mySocket.getInputStream());
            s = dis.readLine();//从 Server 端读入的数据
            System.out.println(s);
            BufferedReader in =
                new BufferedReader(new InputStreamReader(System.in));
            line = in.readLine();
            while(!line.equals("Bye"))
            {
                os.println(line);
                os.flush();
                line = in.readLine();
            }
            os.close();
            dis.close();
            mySocket.close();
        }
        catch (IOException e)
        {
            System.out.println("ClientException"+e);
        }
    }
}
//服务器端程序 ServerTest.java
import java.io.*;
import java.net.*;

```



```
public class ServerTest
{
    public static void main(String[] args)
    {
        try
        {
            boolean flag = true;
            Socket linkSocket = null;
            String inputLine;
            ServerSocket myServer = new ServerSocket(800);
            System.out.println("Server listen on:"+myServer.getLocalPort());
            while(flag)
            {
                linkSocket = myServer.accept();
                DataInputStream dis = new DataInputStream(
                    new BufferedInputStream(linkSocket.getInputStream()));
                //获得从 Client 读入的数据流
                PrintStream ps = new PrintStream(
                    new BufferedOutputStream(linkSocket.getOutputStream()));
                //获得向 Client 输出的数据流
                ps.println("Hello! Welcome connect to my server!");
                ps.flush();//向 Client 端输出信息
                inputLine = dis.readLine();//从 Client 读入信息
                while(!inputLine.equals("Bye"))
                {
                    System.out.println("Client 端输入的信息为: "+inputLine);
                    inputLine = dis.readLine();
                }
                ps.close();
                dis.close();
                linkSocket.close();
                myServer.close();
            }
        }
        catch (IOException e)
        {
            System.out.println("ServerException"+e);
        }
    }
}
```

```

    }
}

```

【例 12-33】例 12-32 的服务器端程序只能为一个客户服务，这里改写例 12-32 的服务器端程序，使之可以为多个客户服务。

//多线程服务器端程序 ServerMulti.java

```

import java.io.*;
import java.net.*;
public class ServerMulti
{
    ServerMulti()
    {
        try
        {
            boolean flag = true;
            Socket links = null;
            ServerSocket myServer = new ServerSocket(800);
            System.out.println("Server listen on:"+myServer.getLocalPort());
            while(flag)
            {
                links = myServer.accept();
                ServiceThread myST = new ServiceThread(links);
                myST.start();
            }
        }
        catch (IOException e)
        {
            System.out.println("ServerException"+e);
        }
    }
    public static void main(String[] args)
    {
        new ServerMulti();
    }
}
class ServiceThread extends Thread
{
    Socket linkSocket;
    String inputLine;

```



```
public ServiceThread(Socket ls)
{
    linkSocket = ls;
}
public void run()
{
    try
    {
        DataInputStream dis = new DataInputStream(
            new BufferedInputStream(linkSocket.getInputStream()));
        PrintStream ps = new PrintStream(
            new BufferedOutputStream(linkSocket.getOutputStream()));
        ps.println("Hello! Welcome connect to my server!");
        ps.flush();           //向 Client 端输出信息
        inputLine = dis.readLine(); //从 Client 端读入信息
        while(!inputLine.equals("Bye"))
        {
            System.out.println("Client 端输入的信息为: "+inputLine);
            inputLine = dis.readLine();
        }
        ps.close();
        dis.close();
        linkSocket.close();
    }
    catch (IOException e)
    {
        System.out.println("ServerException"+e);
    }
}
```



12.16.3 DatagramPacket类和DatagramSocket类

1. DatagramPacket类

构造函数:

```
public DatagramPacket(byte ibuf[], int ilength);
public DatagramPacket(byte ibuf[], int ilength, InetAddress iaddr, int iport);
```

其中, ibuf[]为要接收或发送的字节数组, ilength 为长度, iaddr 为接收者地址, iport

为端口号。

2. DatagramSocket类

构造函数：

```
public DatagramSocket();
//连接到本地主机的任一个可用的端口上

public DatagramSocket(int port);
//在指定端口创建对象

public DatagramSocket(int port, InetAddress localAddr);
//用于在多 IP 地址主机上创建对象
```

三个构造函数都抛出 `IOException` 异常。

数据报接收：

```
public synchronized void receive(DatagramPacket p) throws IOException;
//该方法使程序线程处于堵塞状态，直至收到信息
```

用第一个构造函数创建一个接收数据报的 `DatagramPacket` 对象，包含一个接收数据的空白数据缓冲区，然后在指定或可用的本机端口创建 `DatagramSocket` 对象，调用该 `DatagramSocket` 的 `receive()` 方法，以 `DatagramPacket` 为参数接收数据报。

数据报发送：

```
public void send(DatagramPacket p) throws IOException;
```

用第二个构造函数创建 `DatagramPacket` 对象，包含要发送的数据、发送的目的主机地址和端口号；在指定或可用的本机端口创建 `DatagramSocket` 对象；调用该 `DatagramSocket` 的 `send()` 方法，以 `DatagramPacket` 为参数发送数据报。

【例 12-34】 客户机向服务器发送一字符串，服务器把字符串转换成大写形式，再发送给客户机。

```
//UDP 服务器端程序 UDPServer.java
import java.io.*;
import java.net.*;

class UDPServer
{
    public static void main(String args[]) throws Exception
    {
        DatagramSocket serverSocket = new DatagramSocket(9876);
        byte[] receiveData = new byte[1024];
        byte[] sendData = new byte[1024];
        while(true)
        {
            DatagramPacket receivePacket =
                new DatagramPacket(receiveData, receiveData.length);
            serverSocket.receive(receivePacket);
```



```
String sentence = new String(receivePacket.getData());
InetAddress IPAddress = receivePacket.getAddress();
int port = receivePacket.getPort();
String capitalizedSentence = sentence.toUpperCase();
sendData = capitalizedSentence.getBytes();
DatagramPacket sendPacket =
    new DatagramPacket(sendData, sendData.length, IPAddress, port);
serverSocket.send(sendPacket);
}
}
}
//UDP 客户端程序 UDPClient.java
import java.io.*;
import java.net.*;
class UDPClient
{
    public static void main(String args[]) throws Exception
    {
        BufferedReader inFromUser =
            new BufferedReader(new InputStreamReader(System.in));
        DatagramSocket clientSocket = new DatagramSocket();
        //InetAddress IPAddress = InetAddress.getLocalHost("hostname");
        byte[] sendData = new byte[1024];
        byte[] receiveData = new byte[1024];
        String sentence = inFromUser.readLine();
        sendData = sentence.getBytes();
        DatagramPacket sendPacket = new DatagramPacket(
            sendData,sendData.length,InetAddress.getLocalHost(),9876);
        clientSocket.send(sendPacket);
        DatagramPacket receivePacket =
            new DatagramPacket(receiveData, receiveData.length);
        clientSocket.receive(receivePacket);
        String modifiedSentence = new String(receivePacket.getData());
        System.out.println("FROM SERVER:" + modifiedSentence);
        clientSocket.close();
    }
}
```

12.16.4 URL类和URLConnection类

1. URL类

URL 地址由协议、主机名、路径文件名和端口号 4 个部分组成。例如：

`http://www.tsinghua.edu.cn:80/index.html`

构造函数：

```
public URL(String protocol, String host, int port, String file);
public URL(String protocol, String host, String file)
    throws MalformedURLException;
public URL(String spec) throws MalformedURLException;
public URL(URL contex, String spec) throws MalformedURLException;
```

常用方法：

```
public int getPort();
public String getProtocol();
public String getHost();
public String getFile();
public URLConnection openConnection();
public final InputStream openStream();
public final Object getContent();
```

URL 对象调用 `InputStream openStream()` 方法可以返回一个输入流，该输入流指向 URL 对象所包含的资源。通过该输入流可以将服务器上的资源信息读入到客户端。

【例 12-35】 在一个文本框里输入网址，然后单击“确定”按钮，就可以读取服务器上的资源。由于网速或其他因素，URL 资源读取可能会引起堵塞，因此程序在一个线程中读取 URL 资源，避免堵塞主线程。

```
//URLTest.java
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;
public class URLTest
{
    public static void main(String args[])
    {
        new NetWin();
    }
}
class NetWin extends Frame implements ActionListener,Runnable
```



```
{
    Button button;
    URL url;
    TextField text;
    TextArea area;
    byte b[]=new byte[118];
    Thread thread;
    NetWin()
    {
        text=new TextField(20);
        area=new TextArea(12,12);
        button=new Button("确定");
        button.addActionListener(this);
        thread=new Thread(this);
        Panel p=new Panel();
        p.add(new Label("输入网址:"));
        p.add(text);
        p.add(button);
        add(area,BorderLayout.CENTER);
        add(p,BorderLayout.NORTH);
        setBounds(60,60,360,300);
        setVisible(true);
        validate();
        addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent e)
            {
                System.exit(0);
            }
        });
    }

    public void actionPerformed(ActionEvent e)
    {
        if(!(thread.isAlive()))
            thread=new Thread(this);
        try
        {
```



```

        thread.start();
    }
    catch(Exception ee)
    {
        text.setText("我正在读取"+url);
    }
}
public void run()
{
    try
    {
        int n=-1;
        area.setText(null);
        url=new URL(text.getText().trim());
        //删除字符串首部和尾部的空格
        InputStream in=url.openStream();
        while((n=in.read(b))!=-1)
        {
            String s=new String(b,0,n);
            area.append(s);
        }
    }
    catch(MalformedURLException e1)
    {
        text.setText(""+e1);
        return;
    }
    catch(IOException e1)
    {
        text.setText(""+e1);
        return;
    }
}
}

```

2. URLConnection类

利用给定的 URL 地址创建一个 URL 类对象，调用该对象的 openConnection()方法，就可以返回一个对应于 URL 地址的 URLConnection 对象：

```
URL myUrl=new URL("http://www.tsinghua.edu.cn/")
```



```
URLConnection myUrlConn = myUrl.openConnection();
```

使用 `URLConnection` 可向远方计算机传送信息。



12.17 图形用户界面的设计与实现

通过图形用户界面（Graphics User Interface, GUI），用户和程序之间可以方便地进行交互。Java 的图形用户界面的设计主要有两种方式。

- ◎ **AWT (Abstract Window Toolkit)**: 组件全部是 **HeavyWeight**（重组件），AWT 把显示组件和处理组件事件的工作交给本地组件（同位体）来完成。
- ◎ **Swing**: 组件大部分都是 **LightWeight**（轻组件），轻组件没有同位体，把显示组件和处理组件事件的工作交给相应的 UI 代表来完成。Swing 是架构在 AWT 之上的，没有 AWT 就没有 Swing。

程序员可以根据自己的习惯选择使用 AWT 或 Swing。但是，最好不要二者混用，除显示风格不同之外，还很可能造成层次（Z-Order）错乱。

12.17.1 图形用户界面的构成

图形用户界面由下面 3 种成分构成。

- ◎ **容器**: 容器是用来组织其他界面成分和元素的单元。**Container**（Java 语言定义的一个系统类）的直接或间接子类创建的对象称为容器。
- ◎ **控制组件**: 与容器不同，控制组件是图形用户界面的最小单位之一，它里面一般不再包含其他的成分。
- ◎ **用户自定义成分**。

容器的主要作用和特点如下。

- ◎ 容器有一定的范围，还有一定的位置，这个位置可以是屏幕四角的绝对位置，也可以是相对于其他容器边框的相对位置。
- ◎ 容器通常都有一个背景，这个背景覆盖全部容器，可以是透明的，也可以指定一幅特殊的图案，使界面生动化和个性化。
- ◎ 容器中可以包含其他的许多界面成分和元素。当容器被打开显示时，它上面的所有成分和元素也同时显示出来；当容器被关闭和隐藏时，它所包含的成分和元素也一起被隐藏起来。
- ◎ 容器可以按一定的规则来物理地安排它所包含的元素，如这些元素的相对位置关系、它们的前后排列关系等。
- ◎ 容器可能被包含在其他容器之中。

常用的控制组件有：

- ◎ 标签（**Label**）
- ◎ 按钮（**Button**）
- ◎ 文本编辑区（**TextField, TextArea**）

- ◎ 复选框 (Checkbox)
- ◎ 单选按钮 (CheckboxGroup 或 RadioButton)
- ◎ 下拉列表 (List 或 Choice)

除了标准的图形界面元素外,编程人员还可以根据用户需要设计一些用户自定义的图形界面成分,例如,绘制几何图形,使用标志图案等。用户自定义成分由于不能像标准界面元素一样被系统识别和承认,所以通常只能起到装饰、美化等作用,不能响应用户的动作,不具有交互功能。

AWT 和 Swing 的 Z-order 规则一样:

- ◎ 组件的 Z-order 一定比其容器高,组件一定位于容器上层。
- ◎ 同一个容器的两个组件,越早加入容器者,其 Z-order 越高,位置越上层。

但是如果混合使用 AWT 和 Swing,上述第 2 条规则就不一定了。例如,在某容器内先加入一个 Swing 的组件,再加入一个 AWT 的组件,且这两个组件有重叠的区域,结果却是 AWT 组件出现在 Swing 组件上面。

12.17.2 布局管理

在实际编程中,我们每设计一个窗体,都要往其中添加若干组件。为了管理好这些组件的布局,我们就需要使用布局管理器。Java 在布局管理上采用了容器和布局管理分离的方案。容器只负责将其组件放入其中(使用 add()方法),把组件如何放置的布局管理交给专门的布局管理器类(LayoutManager)来完成,其布局设置语句如下:

```
setLayout(new 布局设计方式());
```

下面,我们将分别介绍 java.awt 包中的 FlowLayout、BorderLayout、CardLayout、GridLayout 布局类和 java.swing.border 包中的 BoxLayout 布局类。

1. FlowLayout 布局

FlowLayout 类创建的对象称做 FlowLayout 型布局。FlowLayout 型布局是 Panel 型容器的默认布局。Panel 及其子类创建的容器对象,如果不专门为其指定布局,则它们的布局就是 FlowLayout 型布局。

一个容器使用这个 FlowLayout 型布局,组件将按照加入的先后顺序从左向右排列,一行排满之后就转到下一行继续从左至右排列。

构造函数:

```
FlowLayout();
```

```
FlowLayout(int align);
```

```
FlowLayout(int align,int hgap, int vgap);
```

align 可取值: FlowLayout.LEFT、FlowLayout.CENTER 和 FlowLayout.RIGHT。

hgap 为横间距,单位为像素;

vgap 为纵间距,单位为像素。

【例 12-36】

```
//FlowExamp.java
```



```
import java.awt.*;

public class FlowExamp
{
    public static void main(String args[])
    {
        WindowFlow win=new WindowFlow("FlowLayout 布局窗口");
    }
}

class WindowFlow extends Frame
{
    WindowFlow(String s)
    {
        super(s);
        FlowLayout flow=new FlowLayout();
        flow.setAlignment(FlowLayout.LEFT);
        flow.setHgap(2);
        flow.setVgap(8);
        setLayout(flow);
        for(int i=1;i<=10;i++)
        {
            Button b=new Button("i am "+i);
            add(b);
        }
        setBounds(100,100,150,120);
        setVisible(true);
    }
}
```

运行结果如图 12-9 所示。



图 12-9 运行结果

2. BorderLayout (边框布局)

BorderLayout 布局是 Window 型容器的默认布局，例如，Frame、Dialog 都是 Window 类的子类，它们的默认布局都是 BorderLayout 布局。如果一个容器使用这种布局，那么容器空间简单地划分为东、西、南、北、中 5 个区域，中间的区域最大。

构造函数：

```
BorderLayout();
//各组件纵、横间距为 0
BorderLayout(int hgap, int vgap);
//hgap 为横间距，vgap 为纵间距，单位为像素
add (组件,位置);
```

位置可取值:

BorderLayout.NORTH[WEST,CENTER,EAST,SOUTH]

【例 12-37】

```
//BorderExamp.java
import java.awt.*;
class BorderExamp
{
    public static void main(String args[])
    {
        Frame win=new Frame("窗体");
        win.setBounds(100,100,300,300);
        win.setVisible(true);
        Button bSouth=new Button("我在南边"),
            bNorth=new Button("我在北边"),
            bEast =new Button("我在东边"),
            bWest =new Button("我在西边");
        TextArea bCenter=new TextArea("我在中心");
        win.add(bNorth,BorderLayout.NORTH);
        win.add(bSouth,BorderLayout.SOUTH);
        win.add(bEast,BorderLayout.EAST);
        win.add(bWest,BorderLayout.WEST);
        win.add(bCenter,BorderLayout.CENTER);
        win.validate();
    }
}
```

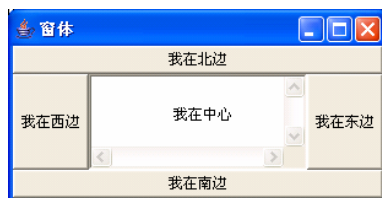


图 12-10 运行结果

运行结果如图 12-10 所示。

3. CardLayout (卡片布局)

使用 CardLayout 的容器可以容纳多个组件,但是实际上在同一时刻,容器只能从这些组件中选出一个来显示,就像一叠“扑克牌”每次只能显示最上面一张一样,这个被显示的组件将占据所有的容器空间,依次排序。采用下面的语句把组件加入容器中:

add(组件名称, 组件);

调用 CardLayout 的方法:

Show(容器名 con,组件名 s); //显示组件

first(con), last(con), next(con), previous(con)和 show(con,s)方法均可使卡片可见。

【例 12-38】

```
//CardExamp.java
import java.awt.*;
import java.awt.event.*;
```



```
public class CardExamp
{
    public static void main(String args[])
    {
        new WinCard();
    }
}

class WinCard extends Frame implements ActionListener
{
    CardLayout mycard;
    Button buttonFirst,buttonLast,buttonNext;
    Panel pCenter;
    WinCard()
    {
        mycard=new CardLayout();
        pCenter=new Panel();
        pCenter.setLayout(mycard);
        buttonFirst=new Button("first");
        buttonLast=new Button("last");
        buttonNext=new Button("next");
        for(int i=1;i<=20;i++)
        {
            pCenter.add("i am"+i,new Button("我是第 "+i+" 个按钮"));
        }
        buttonFirst.addActionListener(this);
        buttonLast.addActionListener(this);
        buttonNext.addActionListener(this);
        Panel pSouth=new Panel();
        pSouth.add(buttonFirst);
        pSouth.add(buttonNext);
        pSouth.add(buttonLast);
        add(pCenter,BorderLayout.CENTER);
        add(pSouth,BorderLayout.SOUTH);
        setBounds(10,10,200,190);
        setVisible(true);
        validate();
    }
    public void actionPerformed(ActionEvent e)
```

```

{
    if(e.getSource()==buttonFirst)
    {
        mycard.first(pCenter);
    }
    else if(e.getSource()==buttonNext)
    {
        mycard.next(pCenter);
    }
    else if(e.getSource()==buttonLast)
    {
        mycard.last(pCenter);
    }
}
}

```

运行结果如图 12-11 所示。

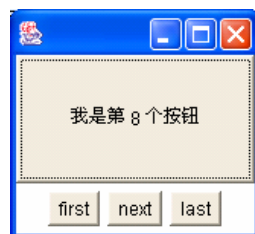


图 12-11 运行结果

4. GridLayout（网格布局）

GridLayout 的布局策略是把容器划分成网格区域，组件就位于这些划分出来的小格中。GridLayout 比较灵活，划分多少网格由程序自由控制，而且组件定位也比较精确。由于 GridLayout 布局中每个网格都是相同大小的，并且强制组件的大小与网格的大小相同，显得很 unnatural。为了克服这个缺点，可以使用容器嵌套策略。其构造函数如下：

Gridout(行数,列数);

Gridout(行数,列数, int hgap, int vgap);

调用 add ()方法按顺序加入组件。若希望某个网格为空，可以为它加入一个空标签：

add(new Label());

【例 12-39】

```

//GridExamp.java
import java.awt.*;
import java.awt.event.*;
public class GridExamp
{
    public static void main(String args[])
    {
        new WinGrid();
    }
}
class WinGrid extends Frame
{
    GridLayout grid;
    WinGrid()

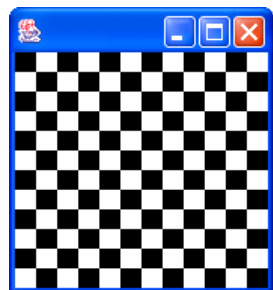
```



```
{
    grid=new GridLayout(12,12);
    setLayout(grid);
    Label label[][]=new Label[12][12];
    for(int i=0;i<12;i++)
    {
        for(int j=0;j<12;j++)
        {
            label[i][j]=new Label();
            if((i+j)%2==0)
                label[i][j].setBackground(Color.black);
            else
                label[i][j].setBackground(Color.white);
            add(label[i][j]);
        }
    }
    setBounds(10,10,260,260);
    setVisible(true);
    validate();
}
```

运行结果如图 12-12 所示。

错误!



5. BoxLayout (盒式布局)

盒式布局的容器将组件排列成一行或一列，这取决于创建盒式布局对象时，是否指定了是行排列还是列排列。使用行（列）型盒式布局的容器将组件排列在一行（列）中，组件按加入的先后顺序从左（上）向右（下）排列。容器的两端是剩余的空间。与 FlowLayout 布局不同的是，使用行型盒式布局的容器只有一行（列），即使组件再多，也不会延伸到下一行（列），这些组件可能会被缩小大小，压缩在这一行（列）中。

【例 12-40】

```
//BoxExamp.java
import javax.swing.*;
import java.awt.*;
import javax.swing.border.*;

public class BoxExamp
{
    public static void main(String args[])
    {
        new WindowBox();
    }
}
```



```

class WindowBox extends Frame
{
    Box baseBox ,boxV1,boxV2;
    WindowBox()
    {
        //得到列型盒式布局的盒式容器
        boxV1=Box.createVerticalBox();
        boxV1.add(new Label("姓名"));
        //垂直支撑
        boxV1.add(Box.createVerticalStrut(8));
        boxV1.add(new Label("email"));
        boxV1.add(Box.createVerticalStrut(8));
        boxV1.add(new Label("职业"));
        boxV2=Box.createVerticalBox();
        boxV2.add(new TextField(12));
        boxV2.add(Box.createVerticalStrut(8));
        boxV2.add(new TextField(12));
        boxV2.add(Box.createVerticalStrut(8));
        boxV2.add(new TextField(12));
        //得到行型盒式布局的盒式容器
        baseBox=Box.createHorizontalBox();
        baseBox.add(boxV1);
        //水平支撑
        baseBox.add(Box.createHorizontalStrut(10));
        baseBox.add(boxV2);
        setLayout(new FlowLayout());
        add(baseBox);
        setBounds(120,125,250,150);
        setVisible(true);
    }
}

```

运行结果如图 12-13 所示。

错误!



12.17.3 组件和事件处理

学习组件除需要了解组件的属性和功能外,一个更重要的方面是学习怎样处理组件上发生的界面事件。在学习处理事件时,必须很好地掌握事件源、监视器、处理事件的接口这三个概念,如图 12-14 所示。通过处理文本框这个具体的组件上的事件,来掌握处理事件的基



本原理。

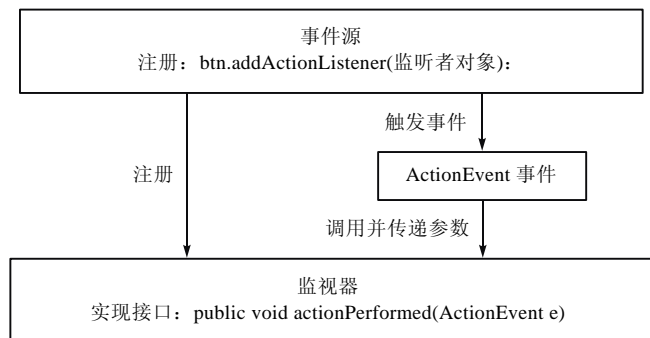


图 12-14 事件处理

事件源：能够产生事件的对象都可以成为事件源，如文本框、按钮、下拉式列表等。也就是说，事件源必须是一个对象，而且这个对象必须是 Java 认为能够发生事件的对象。

监视器：需要一个对象对事件源进行监视，以便对发生的事件做出处理。事件源通过调用相应的方法将某个对象作为自己的监视器。例如，对于文本框，这个方法是：

```
addActionListener(监视器);
```

创建该监视器对象的类必须声明实现相应的接口。

处理事件的接口：在文本框获得输入焦点之后，如果用户按回车键，Java 运行系统就自动用 `ActionEvent` 类创建一个对象，即发生 `ActionEvent` 事件。当事件源发生事件时，监视器自动调用执行被类实现的某个接口方法。

1. `ActionEvent`事件

```
public void addActionListener(ActionListener)
public void actionPerformed(ActionEvent e)
```

相关组件：`TextField`、`Button` 和 `MenuItem`

【例 12-41】以菜单项为例，编写程序如下。

```
import java.awt.*;
import java.awt.event.*;
public class Example1
{
    public static void main(String args[])
    { WindowExit win=new WindowExit(); }
}
class WindowExit extends Frame implements ActionListener
{
    MenuBar menubar;
    Menu menu;
```

```

MenuItem itemExit;
WindowExit()
{
    //创建菜单栏
    menubar=new MenuBar();
    //创建“文件”菜单
    menu=new Menu("文件");
    //创建“退出”菜单项
    itemExit=new MenuItem("退出");
    //设置菜单项快捷键
    itemExit.setShortcut(new MenuShortcut(KeyEvent.VK_E));
    //向菜单添加菜单项
    menu.add(itemExit);
    //向菜单栏添加菜单
    menubar.add(menu);
    //将菜单栏添加到窗口顶端
    setMenuBar(menubar);
    itemExit.addActionListener(this);
    setBounds(100,100,150,150);
    setVisible(true);
    validate();
}
public void actionPerformed(ActionEvent e)
{
    System.exit(0);
}
}

```

运行结果如图 12-15 所示。

2. TextEvent事件

```

public void addTextListener(TextListener)
public void textValueChanged(TextEvent e)

```

相关组件：TextArea

java.awt 包中的类 `TextArea` 类是专门用来建立文本区的，即 `TextArea` 创建的一个对象称做一个文本区。用户可以在文本区中输入多行的文本，如图 12-16 所示。

其常用方法如下：

```
TextArea(int x,int y);
```

使用这个构造方法创建文本区对象，文本框可见行数和列数分别为 `x` 和 `y`。文本区有水平和垂直滚动条。

```
public void setText(String s);
```



错误!



图 12-15 运行结果

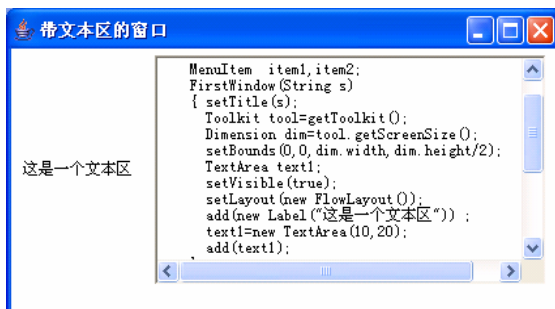


图 12-16 文本区

文本区对象调用该方法可以将文本区中的文本设置为参数 s 指定的文本, 文本区中先前的文本将被清除。

```
public String getText();
```

```
//文本区对象调用该方法可以获取文本区中的文本
```

```
public void append(String s);
```

```
//文本区对象调用该方法可以在文本区中追加文本 .
```

```
addTextListener(TextListener);
```

```
//文本区对象调用该方法可以向文本框中增加文本监视器
```

【例 12-42】

```
import java.awt.*;
```

```
import java.awt.event.*;
```

```
import java.util.*;
```

```
public class Example2
```

```
{
```

```
    public static void main(String args[])
```

```
    { WindowTextArea win=new WindowTextArea(); }
```

```
}
```

```
class WindowTextArea extends Frame
```

```
    implements TextListener,ActionListener
```

```
{
```

```
    TextArea text1,text2;
```

```
    Button buttonClear;
```

```
    WindowTextArea()
```

```
{
```

```
    setLayout(new FlowLayout());
```

```
    text1=new TextArea(6,15);
```

```
    text2=new TextArea(6,15);
```

```
    buttonClear=new Button("清空");
```

```
    add(text1);
```

```

        add(text2);
        add(buttonClear);
        text2.setEditable(false);
        text1.addTextListener(this);
        buttonClear.addActionListener(this);
        setBounds(100,100,350,160);
        setVisible(true);
        validate();
    }

    public void textValueChanged(TextEvent e)
    {
        String s=text1.getText();
        //把一个 StringTokenizer 对象称做一个字符串分析器
        //分析器可以使用 nextToken()方法逐个获取字符串中的语言符号（单词）

        StringTokenizer fenxi=new StringTokenizer(s,"\\n");
        //每当获取到一个语言符号后，字符串分析器中的负责计数的变量的值就自动减 1
        //该计数变量的初始值等于字符串中的单词数目
        int n=fenxi.countTokens();
        String a[]=new String[n];
        for(int i=0;i<=n-1;i++)
        {
            String temp=fenxi.nextToken();
            a[i]=temp;
        }
        Arrays.sort(a);
        text2.setText(null);
        for(int i=0;i<n;i++)
        {   text2.append(a[i]+"\\n"); }
    }

    public void actionPerformed(ActionEvent e)
    {   text1.setText(null); }
}

```

运行结果如图 12-17 所示。

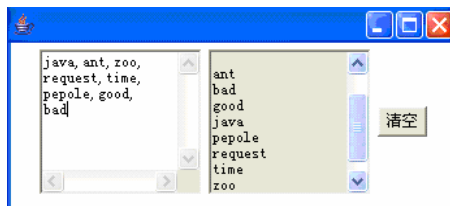


图 12-17 运行结果

3. ItemEvent 事件

```

public void addItemListener(ItemListener)
public void ItemStateChanged (ItemEvent e)

```



相关组件：

Checkbox	//选择框
CheckboxGroup	//多选一选择框
Choice	//下拉列表
List	//滚动列表

【例 12-43】

```
import java.awt.*;
import java.awt.event.*;
public class Example3
{
    public static void main(String args[])
    {
        new WindowBox(); }
class WindowBox extends Frame
{
    Mypanel1 panel1;
    Mypanel2 panel2;
    TextArea text;
    WindowBox()
    {
        text=new TextArea();
        panel1=new Mypanel1(text);
        panel2=new Mypanel2(text);
        add(panel1,BorderLayout.SOUTH);
        add(panel2,BorderLayout.NORTH);
        add(text,BorderLayout.CENTER);
        setSize(200,200);
        setVisible(true);
        validate();
    }
}
class Mypanel1 extends Panel implements ItemListener
{
    Checkbox box1,box2;
    CheckboxGroup sex;
    TextArea text;
    Mypanel1(TextArea text)
    {
```

```

        this.text=text;
        sex=new CheckboxGroup();
        // Checkbox("选择框名",状态,选择框组);
        box1=new Checkbox("男",true,sex);
        box2=new Checkbox("女",false,sex);
        box1.addItemListener(this);
        box2.addItemListener(this);
        add(box1);
        add(box2);
    }

    public void itemStateChanged(ItemEvent e)
    {
        //获取事件源
        Checkbox box=(Checkbox)e.getSource();
        if(box.getState())//得到选择框的布尔状态
        {
            //获取文本区域光标位置
            int n=text.getCaretPosition();
            //插入字符
            text.insert(box.getLabel(),n);
        }
    }
}

class Mypanel2 extends Panel implements ItemListener
{
    Checkbox box1,box2,box3;
    TextArea text;
    Mypanel2(TextArea text)
    {
        this.text=text;
        box1=new Checkbox("张三");
        box2=new Checkbox("李四");
        box3=new Checkbox("王五");
        box1.addItemListener(this);
        box2.addItemListener(this);
        box3.addItemListener(this);
        add(box1);
        add(box2);
    }
}

```



```
        add(box3);
    }
    public void itemStateChanged(ItemEvent e)
    {
        //获取事件源
        Checkbox box=(Checkbox)e.getItemSelectable();
        if(box.getState())
        {
            int n=text.getCaretPosition();
            text.insert(box.getLabel(),n);
        }
    }
}
```

运行结果如图 12-18 所示。

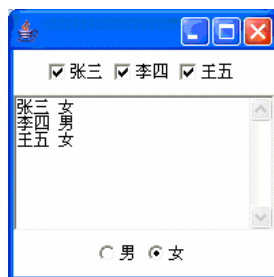


图 12-18 运行结果

4. 窗口事件

Frame 是 Window 的子类。凡是 Window 子类创建的对象都可以发生 WindowEvent 类型事件，即窗口事件。当一个 Frame 窗口被激活、撤销激活、打开、关闭、最小化或撤销最小化时，就会引发窗口事件。获得监视器的方法如下：

```
public void addWindowListener(WindowListener)
```

WindowListener 接口中有 7 个不同的方法：

```
public void windowActivated(WindowEvent e)
```

//当窗口从非激活状态转到激活状态时，窗口的监视器调用该方法

```
public void windowDeactivated(WindowEvent e)
```

//当窗口激活状态转到非激活状态时，窗口的监视器调用该方法

```
public void windowClosing(WindowEvent e)
```

//当窗口正在关闭时，窗口的监视器调用该方法

```
public void windowClosed(WindowEvent e)
```

//当窗口关闭后，窗口的监视器调用该方法

```
public void windowIconified(WindowEvent e)
```

//当窗口图标(最小)化时，窗口的监视器调用该方法

```
public void windowDeiconified(WindowEvent e)
```

//当窗口撤销图标(最小)化时，窗口的监视器调用该方法

```
public void windowOpened(WindowEvent e)
```

//当窗口打开时，窗口的监视器调用该方法

【例 12-44】

```
import java.awt.*;
```

```
import java.awt.event.*;
```

```
public class Example4
```



```

{
    public static void main(String args[])
    {    new MyFrame(); }
}
class MyFrame extends Frame implements WindowListener
{
    TextArea text;
    MyFrame()
    {
        setBounds(100,100,200,300);
        setVisible(true);
        text=new TextArea();
        add(text, BorderLayout.CENTER);
        addWindowListener(this);
        validate();
    }
    public void windowActivated(WindowEvent e)
    {
        text.append("\n 我被激活");
        validate();
    }
    public void windowDeactivated(WindowEvent e)
    {
        text.append("\n 我不是激活状态了");
        setBounds(0,0,400,400);
        validate();
    }
    public void windowClosing(WindowEvent e)
    {
        text.append("\n 窗口正在关闭呢");
        dispose();
    }
    public void windowClosed(WindowEvent e)
    {
        System.out.println("程序结束运行");
        System.exit(0);
    }
    public void windowIconified(WindowEvent e)

```



```

        { text.append("\n 我图标化了"); }
    public void windowDeiconified(WindowEvent e)
    {
        text.append("\n 我撤销图标化");
        setBounds(0,0,400,400);
        validate();
    }
    public void windowOpened(WindowEvent e){ }
}

```

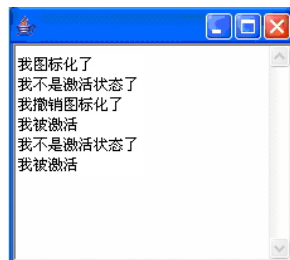


图 12-19 运行结果

运行结果如图 12-19 所示。

当非抽象类实现接口时，即使不准备处理某个方法也需要给出所有方法的实现。当 Java 提供处理事件的接口中的方法多于一个时，就提供一个适配器，如：WindowAdapter 适配器。WindowAdapter 适配器实现了 WindowListener 接口，因此用 WindowAdapter 的子类创建的对象作为监视器，在子类中只需重写需要的方法即可。

【例 12-45】

```

import java.awt.*;
import java.awt.event.*;
public class Example5
{
    public static void main(String args[])
    { new MyFrame("窗口"); }
}
class MyFrame extends Frame
{
    TextArea text;
    Boy police;
    MyFrame(String s)
    {
        super(s);
        police=new Boy(this);
        setBounds(100,100,200,300);
        setVisible(true);
        text=new TextArea();
        add(text, BorderLayout.CENTER);
        addWindowListener(police);
        validate();
    }
}

```

```

class Boy extends WindowAdapter
{
    MyFrame f;
    public Boy(MyFrame f)
    { this.f=f; }
    public void windowActivated(WindowEvent e)
    { f.text.append("\n 我被激活"); }
    public void windowClosing(WindowEvent e)
    { System.exit(0); }
}

```

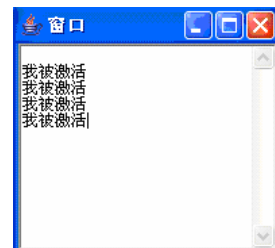


图 12-20 运行结果

运行结果如图 12-20 所示。

5. MouseEvent 事件

```
public void addMouseListener(MouseListener)
```

MouseListener 接口的方法:

```

mousePressed(MouseEvent e)//按下鼠标
mouseReleased(MouseEvent e)//释放鼠标
mouseEntered(MouseEvent e)//鼠标进入容器
mouseExited(MouseEvent e)//鼠标离开容器
mouseClicked(MouseEvent e)//单击鼠标
public void addMouseMotionListener(MouseListener)

```

MouseMotionListener 接口的方法:

```

mouseDragged(MouseEvent e)//负责鼠标拖动事件
mouseMoved(MouseEvent e)//负责鼠标移动事件

```

【例 12-46】

```

import java.awt.*;
import java.awt.event.*;
public class Example6
{
    public static void main(String args[])
    { new WindowMouse(); }
}
class WindowMouse extends Frame implements MouseListener
{
    TextField text;
    Button button;
    TextArea textArea;
    WindowMouse()

```



```
{
    setLayout(new FlowLayout());
    text=new TextField(10);
    text.addMouseListener(this);
    button=new Button("按钮");
    button.addMouseListener(this);
    addMouseListener(this);
    addWindowListener(new WindowAdapter()
    {
        public void windowClosing(WindowEvent e)
        {    System.exit(0); }
    });
    textArea=new TextArea(5,28);
    add(button);
    add(text);
    add(textArea);
    setBounds(100,100,220,120);
    setVisible(true);
}

public void mousePressed(MouseEvent e)
{    textArea.append("\n 鼠标按下,位置:"+"("+e.getX()+","+e.getY()+")"); }

public void mouseReleased(MouseEvent e) {}

public void mouseEntered(MouseEvent e) {}

public void mouseExited(MouseEvent e) {}

public void mouseClicked(MouseEvent e)
{
    if(e.getClickCount()>=2)
        textArea.setText("鼠标连击, 位置:"+"("+e.getX()+","+e.getY()+")");
}
}
```

运行结果如图 12-21 所示。

6. KeyEvent事件

```
public void addKeyListener(KeyListener)
```

KeyListener 接口:

```
keyPressed(KeyEvent e)//按下键盘
```

```
mouseReleased(KeyEvent e)//释放键盘
```

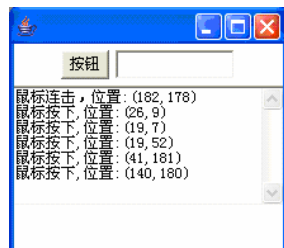


图 12-21 运行结果

mouseTyped(KeyEvent e)//按下并释放键盘

【例 12-47】

```
//Example7.java
import java.awt.*;
import java.awt.event.*;
public class Example7
{
    public static void main(String args[])
    {
        Win win=new Win(); }
}
class Win extends Frame implements KeyListener
{
    Button b[]=new Button[8];
    int x,y;
    Win()
    {
        setLayout(new FlowLayout());
        for(int i=0;i<8;i++)
        {
            b[i]=new Button(""+i);
            b[i].addKeyListener(this);
            add(b[i]);
        }
        addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent e)
            { System.exit(0); }
        })
        setBounds(10,10,300,300);
        setVisible(true);
        validate();
    }
    public void keyPressed(KeyEvent e)
    {
        Button button=(Button)e.getSource();
        x=button.getBounds().x;
        y=button.getBounds().y;
```



```
if(e.getKeyCode()==KeyEvent.VK_UP)
{
    y=y-2;
    if(y<=0) y=0;
    button.setLocation(x,y);
}
else if(e.getKeyCode()==KeyEvent.VK_DOWN)
{
    y=y+2;
    if(y>=300) y=300;
    button.setLocation(x,y);
}
else if(e.getKeyCode()==KeyEvent.VK_LEFT)
{
    x=x-2;
    if(x<=0) x=0;
    button.setLocation(x,y);
}
else if(e.getKeyCode()==KeyEvent.VK_RIGHT)
{
    x=x+2;
    if(x>=300) x=300;
    button.setLocation(x,y);
}
}

public void keyTyped(KeyEvent e) {}
public void keyReleased(KeyEvent e) {}
}
```

运行结果如图 12-22 所示。

错误!

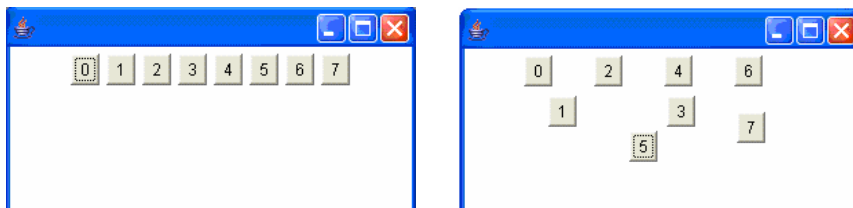


图 12-22 运行结果

7. Dialog

Dialog 和 Frame 都是 Window 的子类，但 Dialog 没有添加菜单的功能，且对话框需要

依赖某个窗口或组件。当它所依赖的窗口或组件消失时，对话框也将消失；而当它所依赖的窗口或组件可见时，对话框又会自动恢复显示。对话框分为无模式和有模式两种。

有模式对话框：处于激活状态时，会堵塞其他线程的执行，直到该对话框不见为止。

无模式对话框：处于激活状态时，不会堵塞其他线程的执行。

(1) 文件对话框

`FileDialog` 是 `Dialog` 类的子类，它创建的对象称为文件话框。文件对话框是一个用于打开文件或保存文件的有模式对话框。打开文件话框如图 12-23 所示。文件对话框必须依附于一个 `Frame` 对象。

`FileDialog` 类具有下列主要方法。

`FileDialog(Frame f, String s, int mode)`：构造方法。参数 `f` 是创建的对话框所依附的窗口对象，`s` 是对话框的名字，`mode` 的取值为 `FileDialog.LOAD` 或 `FileDialog.SAVE`，决定对话框是打开文件模型还是保存文件模型。

`public String getDirectory()`：获取当前文件对话框中显示的文件的所属目录。

`public String getFile()`：获取当前文件对话框中显示的文件的字符串表示。如果不存在，则 `Null`。

(2) 消息对话框

消息对话框是有模式对话框。在进行一个重要的操作动作之前，最好能弹出一个消息对话框，如图 12-24 所示。`JOptionPane` 类的静态方法：

```
public static void showMessageDialog(Component parentComponent,
String message, String title, int messageType)
```

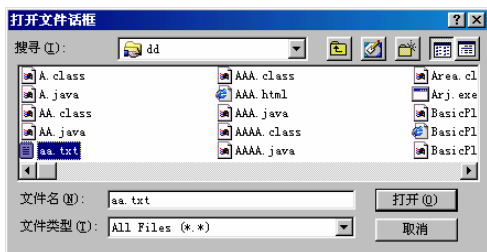


图 12-23 打开文件对话框

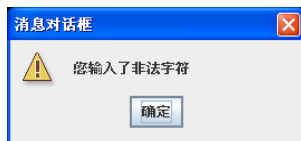


图 12-24 消息对话框

`messageType` 有效值如下：

`JOptionPane.INFORMATION_MESSAGE`

`JOptionPane.WARNING_MESSAGE`

`JOptionPane.ERROR_MESSAGE`

`JOptionPane.QUESTION_MESSAGE`

`JOptionPane.PLAIN_MESSAGE`

这些值可以确定对话框的外观。

(3) 确认对话框

确认对话框是有模式对话框，如图 12-25 所示。`JOptionPane` 类的静态方法：

```
public static int showConfirmDialog(Component parentComponent,
```



Object message, String title, int optionType)

optionType 有效值如下:

JOptionPane.YES_NO_OPTION

JOptionPane.YES_NO_CANCEL_OPTION

JOptionPane.OK_CANCEL_OPTION

(4) 颜色对话框 (调色板)

javax.swing 包中的 JColorChooser 类的静态方法:

public static Color showDialog(Component component, String title, Color initialColor)

参数 component 指定对话框所依赖的组件; title 指定对话框的标题; initialColor 指定对话框返回的初始颜色, 即对话框消失后, 返回的默认值。颜色对话框可根据用户在颜色对话框中选择的颜色返回一个颜色对象, 如图 12-26 所示。

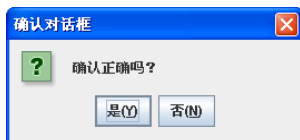


图 12-25 确认对话框



图 12-26 调色板



12.17.4 Java Swing 基础

javax.swing 包为用户提供了更加丰富的、功能强大的组件, 称为 Swing 组件, 其中大部分组件是轻组件, 没有同位体。

Swing 组件的轻组件在设计上与 AWT 完全不同。轻组件把与显示组件有关的许多工作和处理组件事件的工作交给相应的 UI 代表来完成。

UI 代表是用 Java 语言编写的类, 这些类被增加到 Java 的运行环境中, 因此组件的外观不依赖于平台, 不仅在不同平台上的外观是相同的, 而且比重组件有更高的性能。

1. 几个重要的类

javax.swing 包中有 4 个最重要的类: JComponent 类及其子类都是轻组件; 而 JFrame、JApplet、JDialog 都是重组件, 即有同位体的组件, 它们可以与操作系统交互信息。轻组件必须要在这些重量容器中绘制自己。

(1) JComponent 类

JComponent 类是所有轻组件的父类 (就像 Component 类是所有重量组件的父类一样)。

JComponent 的几个代表子类:

◎ JLabel 负责创建标签;

◎ JMenu 负责创建菜单对象;

◎ JMenuItem 负责创建菜单项对象;

- ◎ JPanel 负责创建面板对象;
- ◎ JPasswordField 负责创建口令文本框对象;
- ◎ JPopupMenu 负责创建弹出式菜单;
- ◎ JProgressBar 负责创建进程条;
- ◎ JRadioButton 负责创建单选按钮;
- ◎ JScrollBar 负责创建滚动条;
- ◎ JScrollPane 负责创建滚动窗格;
- ◎ JSlider 负责创建滑动条;
- ◎ JSplitPane 负责创建拆分窗格。
- ◎ JTable 负责创建表格。

(2) JFrame 类

javax.swing 包中的 JFrame 类是 java.awt 包中 Frame 类的子类。因此, JFrame 类及其子类创建的对象是窗体。由于 Frame 是重容器, 因此 JFrame 类或子类创建的对象(窗体)也是重容器。

(3) JApplet 类

JApplet 类用来建立 Java 小应用程序。JApplet 类是 javax.swing 包中的类, 它还是 java.applet 包中的 Applet 类的子类, 因此 JApplet 对象也是一个重容器。

(4) JDialog 类

JDialog 是 java.awt 包中 Dialog 类的子类。JDialog 类或子类创建的对象也是重量容器, 该对象必须依附一个 JFrame 对象。

2. 中间容器

有一些经常用来添加组件的轻容器, 相对于底层重容器而言, 我们习惯上称这些轻容器为中间容器。

(1) JPanel 面板

我们常使用 JPanel 创建一个面板, 再向这个面板添加组件, 然后把这个面板添加到底层容器或其他中间容器中。JPanel 面板的默认布局是 FlowLayout 布局。

(2) 滚动窗口 JScrollPane

我们可以把一个组件放到一个滚动窗口中, 然后通过滚动条来观察这些组件。

例如, JTextArea 不自带滚动条(这一点与重组件 TextArea 不同), 因此, 我们需要把文本区放到一个滚动窗口中。例如:

```
JScrollPane scroll=new JScrollPane(new JTextArea());
```

(3) 拆分窗口 JSplitPane

拆分窗口就是被分成两部分的窗口。

拆分窗口有两种类型: 水平拆分和垂直拆分。

水平拆分窗口用一条拆分线把窗口分成左、右两部分, 左面放一个组件, 右面放一个组件, 拆分线可以水平移动。

垂直拆分窗口用一条拆分线把窗口分成上、下两部分, 上面放一个组件, 下面放一个组件, 拆分线可以垂直移动。



3. 各种组件

① 按钮(JButton): JButton 类负责创建按钮对象。与重组件按钮 Button 相比, JButton 按钮具有更加丰富的外观。

② 标签(JLabel): JLabel 类负责创建标签对象。与重组件标签 Label 相比, JLabel 标签具有更加丰富的外观, 例如, 可以创建带图标 的标签。

③ 复选框(JCheckBox): JCheckBox 类负责创建复选框对象。与重组件复选框 CheckBox 相比, JCheckBox 复选框的名字可以是字符串, 而且它的样子可以是一个图标。

④ 单选按钮(JRadioButton): 单选按钮和复选框很类似, 所不同的是, 在若干个复选框中可以同时选中多个, 而一组单选按钮在同一时刻只能有一个被选中。

⑤ 下拉列表(JComboBox): 用户可以在下拉列表看到第一个选项和它旁边的箭头按钮。当用户单击箭头按钮时, 对于下拉式列表事件源, 可以发生 ItemEvent 事件。

⑥ 文本框(JTextField)、密码框(JPasswordField)、文本区(JTextArea): JTextField、JTextArea 和重组件的文本框、文本区类似, 分别用于显示单行文本和多行文本。它们的实例方法也和重组件的文本框、文本区相同。例如, 可以使用 setText(string)设置文本, getText()获取文本。需要注意的是, 密码框 JPasswordField 有一个特殊方法: char[] getPassword()。

⑦ 文件选择器(JFileChooser): 文件选择器是一个从文件系统中进行文件选择的界面。文件选择器事实上并不能打开或保存文件, 它们只能替用户得到要打开或保存的文件对象, 要想真正实现打开或保存, 还得使用输入流、输出流。

⑧ 进度条: 使用 JProgressBar 类创建进度条组件。该组件能用一种颜色动态地填充自己, 以便显示某任务完成的百分比。

⑨ 表格(JTable): 表格组件以行和列的形式显示数据, 允许对表格中的数据进行编辑。表格的模型功能强大、灵活并易于执行。



图 12-27 树型结构

⑩ 树(JTree): 一个 JTree 类对象提供了一个用树型结构分层显示数据的视图。树中最基本的对象叫做结点, 它表示在给定层次结构中的数据项。树以垂直方式显示数据, 每行显示一个结点。树中只有一个根结点, 所有其他结点从这里引出, 如图 12-27 所示。除根结点外, 其他结点分为两类: 一类是带子结点的分支结点; 另一类是不带子结点的叶结点。

每一个结点都关联着一个描述该结点的文本标签和图像图标。文本标签是结点的字符串表示, 图标指明该结点是否为叶结点。

习题 12

12.1 根据要求编程实现复数类 ComplexNumber。

(1) 复数类 ComplexNumber 的属性

realPart 为实部, imageinPart 为虚部。

(2) 复数类 ComplexNumber 的方法

ComplexNumber(): 构造函数，将实部和虚部置零。

ComplexNumber(double r,double I): 构造函数，将实部和虚部分别置为 *r* 和 *i*。

getRealPart(): 获得复数对象的实部。

getImageinPart(): 获得复数对象的虚部。

complexAdd(ComplexNumber c) : 当前复数对象与形式参数复数对象相加，所得结果为复数对象，返回调用者。

complexMulti((ComplexNumber c) : 当前复数对象与形式参数复数对象相乘，所得结果为复数对象，返回调用者。

toString() : 把当前复数对象，以 *a+bi* 的字符串形式组合起来，*a* 为实部，*b* 为虚部。

12.2 要求把习题 1 的类打包到 **complex** 中。实现类 **ImpComplex**，使用包 **complex** 中的类，即要求用 **import** 语句引入该包。

12.3 采用套接字编程方式，编写一个简单的字符界面的聊天程序。

参 考 文 献

- [1] Roger S Pressman. Software Engineering:A Practitioner's Approach. 5th ed. McGraw-Hill Companies, Inc., 2001.
- [2] Shari Lawrence Pfeeger. Software Engineering:Theory and Practice (影印版). 2nd ed. 北京: 高等教育出版社, 2001.
- [3] Leszek A Maciaszek. Requirements Analysis and System Design: Developing Information Systems with UML. Pearson Education Limited, 2001.
- [4] Rumbaugh J, Jacobson I, Booch G. The Unified Modeling Language Reference Manual. Addison-Wesley, 1999.
- [5] Frank Buschmann. Pattern-Oriented Software Architecture Volumn1:A System of Patterns (影印版). 北京: 机械工业出版社, 2003.
- [6] John D McGregor, David A Sykes. A Practical Guide to Testing Object-Oriented Software (影印版). 北京: 机械工业出版社, 2002.
- [7] 许家珩, 曾翎, 彭德中. 软件工程——理论与实践. 北京: 高等教育出版社, 2004.
- [8] 齐治昌, 谭庆平, 宁洪. 软件工程 (第二版). 北京: 高等教育出版社, 2004.
- [9] 邵维忠, 杨芙清. 面向对象的系统分析. 北京: 清华大学出版社, 1998.
- [10] 吴际, 金茂忠. UML 面向对象分析. 北京: 北京航空航天大学出版社, 2002.
- [11] 卡耐基梅隆大学软件工程研究所. 能力成熟度模型 (CMM): 软件过程改进指南. 刘孟仁等. 北京: 电子工业出版社, 2001.
- [12] 徐仁佐. 软件工程. 武汉: 华中科技大学出版社, 2000.
- [13] Carma McClure. 软件复用技术. 北京: 机械工业出版社, 2003.
- [14] 张敬等. 软件工程教程. 北京: 北京航空航天大学出版社, 2002.
- [15] 朱三元, 钱乐秋, 宿为民. 软件工程技术概论. 北京: 科学出版社, 2002.
- [16] 刘润东. UML 对象设计与编程. 北京: 北京希望电子出版社, 2001.
- [17] Wendy Boggs 等. UML 与 Rational Rose 2002 从入门到精通. 北京: 电子工业出版社, 2002.
- [18] 郑人杰, 殷人昆, 陶永. 实用软件工程. 北京: 清华大学出版社, 1997.
- [19] Meilir Page-Jones. UML 面向对象设计基础. 北京: 人民邮电出版社, 1999.
- [20] 潘锦平, 施小英, 姚天昉. 软件系统开发技术. 西安: 西安电子科技大学出版社, 1997.
- [21] Bjarne Stroustrup. C++程序设计语言. 麦中凡, 董长忠. 北京: 清华大学出版社, 1988.
- [22] 王博. 面向对象的建模、设计技术与方法. 北京: 北京希望电脑公司, 1992.
- [23] Mullin M. Object_Oriented Program Design: with Examples in C++. Addison_Wesley, 1989.
- [24] Annl. Winblad, Sannmuel D Edwards, David R King. Object_Oriented Software. Addsdion_Wesley, 1992.
- [25] Andrew Koing. Templates as Interface. Joop, 1991.

- [26] Beck, Kent. A Laboratory for Teaching Object_oriented Think. OOPSLA-89 Proceeding.
- [27] 张松梅. C++语言教程. 成都: 电子科技大学出版社, 1993.
- [28] Bruce Eckel. C++编程思想. 刘宗田等译. 北京: 机械工业出版社, 2001.
- [29] Scott Meyers. Effective C++. 武汉: 华中科技大学出版社, 2001.
- [30] Scott Meyers. More Effective C++(2nd Edition). 北京: 中国电力出版社, 2003.
- [31] Bjarne Stroustrup. The C++ Programming Language(Special Edition). 北京: 机械工业出版社, 2002.
- [32] Stanley B, Lippman Barbara E, Moo Josée LaJoie. C++ Primer. 北京: 人民邮电出版社, 2006.
- [33] 陈文字, 白忠建, 戴波. 面向对象程序设计语言 C++ (第 2 版). 北京: 机械工业出版社, 2008.
- [34] 耿祥义, 张跃平. Java 2 实用教程 (第三版) 北京: 清华大学出版社, 2006.
- [35] Bruce Eckel. Java 编程思想 (第四版). 陈昊鹏. 北京: 机械工业出版社, 2007.
- [36] H M Deitel, P J Deitel. Java 程序设计教程 (第 5 版). 施平安, 施惠琼, 柳赐佳. 北京: 清华大学出版社, 2004.
- [37] Qusay H, Mahmoud. Java 分布式程序设计. 欧阳光, 安锦. 北京: 国防工业出版社, 2002.

反侵权盗版声明

电子工业出版社依法对本作品享有专有出版权。任何未经权利人书面许可，复制、销售或通过信息网络传播本作品的行为；歪曲、篡改、剽窃本作品的行为，均违反《中华人民共和国著作权法》，其行为人应承担相应的民事责任和行政责任，构成犯罪的，将被依法追究刑事责任。

为了维护市场秩序，保护权利人的合法权益，我社将依法查处和打击侵权盗版的单位和个人。欢迎社会各界人士积极举报侵权盗版行为，本社将奖励举报有功人员，并保证举报人的信息不被泄露。

举报电话：(010) 88254396; (010) 88258888

传 真：(010) 88254397

E-mail: dbqq@phei.com.cn

通信地址：北京市万寿路 173 信箱

电子工业出版社总编办公室

邮 编：100036